Oracle DBA教程

——从基础到实践

林树泽 苏志同 孔 浩 编著

DVD 超过11小时
多媒体视频讲解
近1000个演示范例

技术要点

- 汇集作者学习与维护Oracle数据库的经验，以清晰的范例详解Oracle数据库管理与维护技术
- 涵盖OCP和OCA考试的Oracle数据库管理的内容，供Oracle OCA或OCP认证考试者阅读
- 基于Oracle 11g新版本，同时也适用于Oracle 9i和Oracle 10g

QQ答疑：848120253

清华大学出版社

Oracle DBA教程

——从基础到实践

林树泽 苏志同 孔 浩 编著

清华大学出版社
北 京

内 容 简 介

Oracle 数据库是一个应用广泛且优秀的关系数据库管理系统。本书全面、详细地讲解了 Oracle 数据库开发和管理技术，是学习 Oracle 数据库管理的一本实用教材。

全书共分 29 章，通过近 1000 个范例详尽讲解了 Oracle 数据库安装与卸载、各种数据库对象、PL/SQL 语言、数据库备份与恢复、用户与系统管理以及数据库性能优化等技术。书中每章的内容不但概念清晰、操作步骤明了、示例丰富，而且更侧重于满足实际工作的需要。

本书适用于想进入 Oracle 数据库领域的初学者，同时也可以满足中级读者或初学者继续深入学习的要求。书中的内容完全覆盖了 OCP 考试中数据库管理的知识点，所以同样适用于参加 OCP 或 OCA 考试的读者（Oracle 数据库管理 I）。

图书在版编目（CIP）数据

Oracle DBA 教程：从基础到实践/林树泽，苏志同，孔浩编著—北京：清华大学出版社，2010.6

ISBN 978-7-302-22503-4

I. ①O…　II. ①林…②苏…③孔…　III. ①关系数据库—数据库管理系统，Oracle—教材　IV. ①TP311.138

中国版本图书馆 CIP 数据核字（2010）第 064166 号

责任编辑： 夏非彼　张　楠
责任校对： 闫秀华
责任印制：
出版发行： 清华大学出版社　　**地　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：** 100084
社 总 机： 010-62770175　　**邮　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销： 全国新华书店
开　本： 190×260　　**印　张：** 35.5　　**字　数：** 910 千字
附光盘 1 张
版　次：　　**印　次：**
印　数：
定　价：

产品编号：

Oracle DBA教程——从基础到实践

[多媒体光盘使用说明]



光盘内容

• 包括近655分钟的多媒体语音视频讲解，涵盖本书的29章内容。

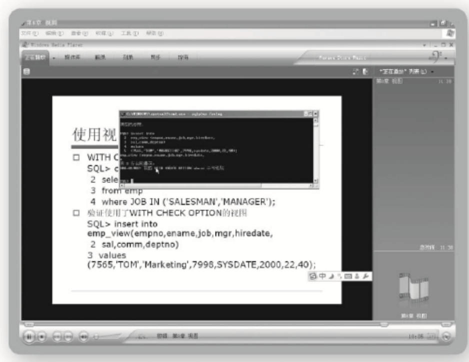
视频课程



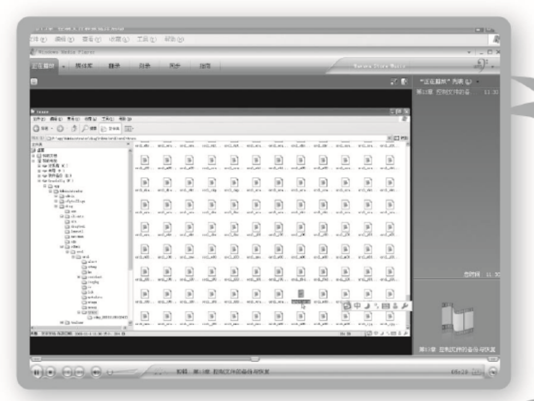
部分视频教学播放界面



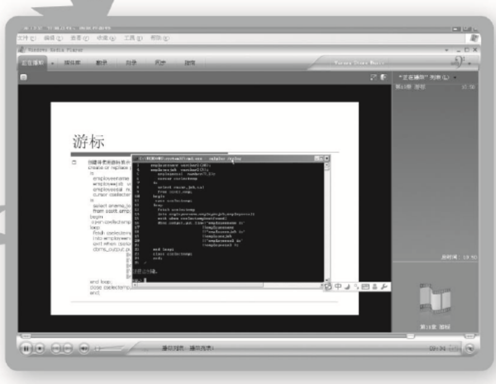
安装数据库



视图



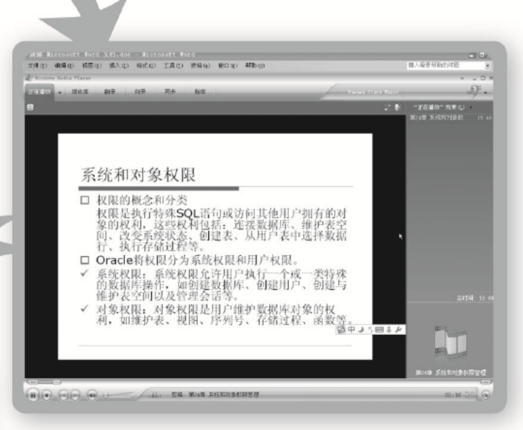
控制文件的备份与恢复



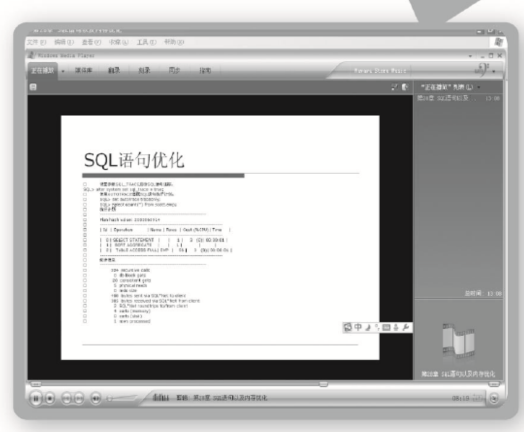
游标



序列号



系统和对象权限



SQL语句优化

前　　言

Oracle 数据库是一个十分优秀的关系数据库管理系统，广泛应用于社会各个领域，如银行、海关、税务、安全部门、经贸部门、保险、金融、航空以及电子商务等，所以从事 Oracle 数据库开发和系统管理、维护工作具有很好的职业前景。

本书共分 29 章，内容包括数据库安装与卸载、数据库对象介绍、PL/SQL 语言、数据库备份与恢复以及数据库性能优化等。

- 第 1 章讲述在 Windows 平台上如何安装和卸载 Oracle 数据库，为以后的学习建立软件平台。对于初学者，推荐使用 Windows 作为学习平台，减少读者由于缺少 Linux 系统基础而增加学习难度。
- 第 2 章介绍了 Oracle 数据库体系结构，在 Oracle 数据库的发展过程中，数据库体系结构是变化最小的，从 Oracle 9i 开始 Oracle 基本保持了相同的系统结构，读者要认真学习该章的每一个数据库组件，如 SGA、各内存组件作用及后台管理进程职责，对于今后深入学习 Oracle 数据库管理十分重要。
- 第 3 章针对如何使用各种 SQL*Plus 指令进行了详细介绍，SQL*Plus 工具是 DBA 维护数据库最古老也是最有效的，它操作简洁，而且工作稳定，是 DBA 必须掌握的数据库管理工具。
- 第 4 章介绍了 SQL 语言的分类、书写规范、常用的 SQL 函数以及 DML 操作方法。
- 第 5 章介绍了如何创建数据库。
- 第 6 章介绍了表的结构以及如何创建、使用和维护表。
- 第 7 章讲解了数据字典的内容、作用、分类以及使用。
- 第 8 章介绍了引入视图的原因，以及如何创建、使用和维护视图。
- 第 9 章详细介绍了事务的特点和事务控制。
- 第 10 章介绍了简单查询、条件查询、子查询以及连接查询。
- 第 11 章介绍了索引和约束的概念，分析了几种常见索引和约束类型。
- 第 12 章分析了数据库启动与关闭的过程，以及涉及到的各种文件。
- 第 13 章分析了数据库启动和控制文件的关系，以及如何维护控制文件。
- 第 14 章介绍了 Oracle 实例及参数文件，说明启动数据库实例和参数文件的关系。
- 第 15 章介绍了表空间的分类、创建方式和表空间类型对应的数据文件的维护。
- 第 16 章深入探讨了 Oracle 引入重做日志的作用以及如何使用、维护和管理重做日志。
- 第 17 章讲述了 Oracle 引入还原段的作用、分类和还原表空间的维护。
- 第 18 章介绍了 PL/SQL 语言的各种流程控制方法，对于创建如触发器、存储过程和函数等数据库对象建立了基础。
- 第 19 章介绍什么是存储过程、函数和游标，以及如何使用 PL/SQL 语言创建数据库对象。
- 第 20 章介绍了触发器的作用、分类和语法格式等，详细说明了编写方法及维护方法。

- 第 21 章介绍了 Oracle 数据库中序列号和同义词的概念、作用及如何创建和维护数据库对象。
- 第 22 章讲述了用户管理、概要文件和资源文件的维护。
- 第 23 章介绍了 Oracle 数据库如何设计和管理角色。
- 第 24 章介绍了 Oracle 数据库中权限的分类，即系统权限、对象权限、授予和回收各种权限。
- 第 25 章介绍使用 EXP 及数据泵完成数据库备份的各种方法和实施步骤，并给出对应的数据库恢复方法。
- 第 26 章详细介绍了 RMAN 的系统结构以及组成，介绍了如何使用 RMAN 进行数据库备份与恢复，通过该章读者可以充分理解 RMAN 的强大功能，轻松地实现数据库的备份与恢复。
- 第 27 章给出实施优化的思路，如从等待事件确定优化目标、迭代地实施优化措施等。
- 第 28 章介绍了书写高效的 SQL 语句准则，以及如何使用工具对 SQL 的执行过程进行分析，对不同的内存组件优化给出具体的指导原则和详细措施。
- 第 29 章详细介绍了如何优化 I/O，给出优化 I/O 的方法，如优化表空间、数据文件、表和索引等，并给出了在不同平台上实施系统级优化的方法。

本书涵盖了 Oracle 11g 数据库开发和管理的全部基础知识，特点主要体现在以下几个方面。

- 全书采用循序渐进的讲解方式，示例丰富，注释清晰，适合初、中级读者学习 Oracle 数据库管理的基础知识，提高维护 Oracle 数据库的实际工作能力。
- 结合笔者的学习和维护 Oracle 数据库的经验，深入浅出地介绍了 Oracle 数据库管理、维护所需要的各个方面的知识，对每章讲解中的关键部分还特别指出初学者的注意事项。
- 尽可能采用浅显易懂的示例进行讲解，并且都使用 Oracle 数据库安装时的默认安装用户 SCOTT 的表来实现各种 SQL 操作和相关概念的讲解，这样读者就不必再创建大量的表，从而减少读者的学习难度和操作复杂性，对于初学者尤为适用。
- 本书在充分介绍 Oracle 11g 数据库的各方面知识外，也讲述了数据库管理员最重要的工作内容，即数据库的备份与恢复，以及数据库性能优化，使得读者在学习 Oracle 数据库基础知识后，更加全面地综合运用基础知识，提高实际的数据库维护能力。
- 书中的示例虽然是以 Oracle 11g 版本为基础而编写，强调了 Oracle 11g 的新特性，但绝大多数的实例都可以在 Oracle 9i 和 10g 上运行，如果读者的计算机硬件资源不足，如内存过低无法安装 Oracle 11g，同样可以学习 Oracle 数据库管理的基础知识。

本书涵盖的知识点基本包括 OCP 和 OCA 考试的 Oracle 数据库管理 I 的内容，对于要参加 Oracle OCA 或 Oracle OCP 认证考试的读者，本书也是一本很好的参考书。

全书主要由林树泽、苏志同、孔浩执笔。此外参与图书编写和视频制作的还有贾东永、李华、王林、赵兵、孙明、李志国、陈晨、冯慧、徐红、杨小庆、魏刚、吴文林、周建国、张建、刘海涛、姚琳、何武、许小荣和林建新等。由于时间仓促，加之水平有限，书中不足之处在所难免，敬请读者批评指正。

编者
2010.6

目　　录

第 1 章

数据库的安装与卸载

在学习 Oracle 数据库之前首先的任务是安装数据库软件，并学会使用基本的数据库管理工具访问数据库。在 Oracle 的不同数据库版本中，数据库软件对于计算机硬件的要求不同，基本趋势是版本越高对计算机硬件要求越高，数据库管理越趋于自动化，主要体现在对内存和 CPU 的要求上。本章我们主要介绍安装数据库软件的操作系统需求以及安装过程，最后介绍如何删除数据库软件。

1.1 安装 Oracle 数据库

数据库软件是运行在操作系统上的，由于它要消耗操作系统的各种资源，如内存、CPU 以及 I/O 等，所以在安装 Oracle 数据库软件之前最好要阅读相关随机文档，了解该软件对于操作系统的要求，然后再安装数据库软件。

1.1.1 系统需求

Oracle 数据库软件可以安装在 Windows NT、Windows 2000 Server、Windows XP 等操作系统上，当然 Oracle 数据库软件也可以安装在 Linux 平台的计算机上。不同版本的 Oracle 数据库软件对于系统硬件（主要是内存）的要求如下所示。

- Oracle 9i 要求至少 256M 内存，但最好在 512M 以上。
- Oracle 10g 要求内存至少 512M，但是最好有 1G 内存。
- Oracle 11g 要求有 2G 内存，当然越高越好。

笔者的计算机是 2G 内存，操作系统为 Windows XP，欲安装 Oracle 11g Release 11.1.0.6.0 版本的数据库软件，下面给出具体的安装过程。

1.1.2 安装过程

本节需要在 Windows XP 系统上安装 Oracle11g Release 11.1.0.6.0，安装过程比较简单，读者需要注意一些参数的设置以及参数含义。

01 对于数据库软件而言，如果出于学习的目的，可以到 Oracle 的官方网站下载，笔者下载的 Oracle 11g 数据库软件包的名称为 win32_11gR1_database_1013（1）.zip，使用压缩工具打开该压缩软件，如图 1-1 所示。

02 单击其中的 Setup.exe 程序即可启动 Oracle 数据库软件的安装过程，此时首先会压缩数据库软件，当解压缩完毕后进入“选择要安装的产品”对话框，如图 1-2 所示。

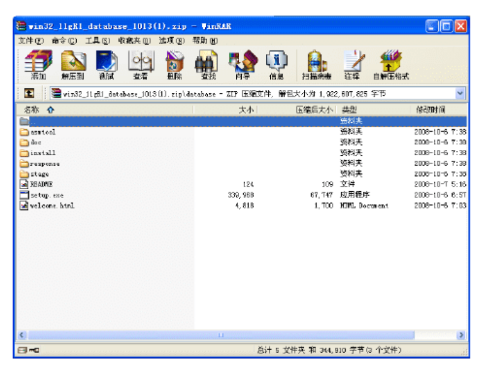


图 1-1　数据库软件的压缩包内容

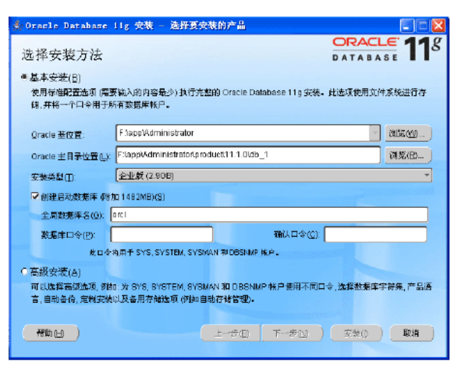


图 1-2　系统检查配置

03 在图 1-2 中需要注意几个参数，第一个是 Oracle 主目录的位置，这里需要读者自己选择在哪个目录下安装 Oracle 数据库软件，安装目录不要有中文目录名；第二个是全局数据库名，该参数的默认值为 orcl，读者可以自己设置；最后一个是数据库口令，注意该参数下面的一行提示“此口令将用于：SYS，YSTEM，SMAN 和 DBSNMP 账户”，笔者的计算机上设置的 Oracle 主目录为 F:\app\Administrator\product\11.1.0\db_1，全局数据库名为 orcl，而数据库口令为 oracle（输入口令时无法明文显示），如图 1-3 所示。

04 单击图 1-3 中的“下一步”按钮，弹出如图 1-4 所示的“准备安装”对话框，该图是临时的，很快会弹出如图 1-5 所示的“准备安装数据库软件”对话框。

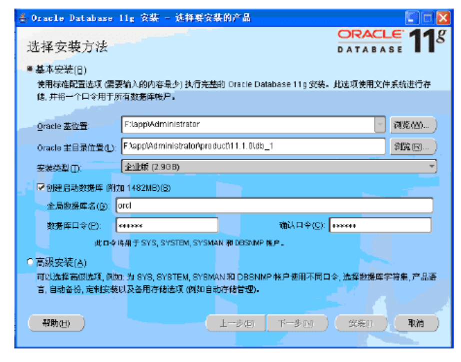


图 1-3　安装数据库



图 1-4　设置数据库安装参数

05 在图 1-5 中，此时执行安装数据库需要的一些先决条件，如网络设置、检查 PATH 环境变量等，在检查完毕后，在“状态”列值中会显示相关检查对象的结果，然后单击图 1-5 中的“下一步”按钮，弹出如图 1-6 所示的“警告”对话框，说明某些条件检查未通过，此时如果是与网络设置相关的警告可以不考虑，直接单击“下一步”按钮进行如图 1-7 所示的相关性处理。



图 1-5　检查安装数据库的条件



图 1-6　安装数据库的条件检查

06 图 1-7 中的相关性检查执行速度很快，完成这个步骤后，单击“下一步”按钮，进入如图 1-8 所示的“概要”对话框，概要信息包括全局设置、产品语言、空间要求和新安装组件，这些信息很直观，这里不做过多解释。单击“安装”按钮，弹出如图 1-9 所示的“安装”对话框。



图 1-7　分析相关性处理



图 1-8　安装数据库的概要信息

07 在图 1-9 中，安装程序根据前面的数据库检查以及参数设置完成数据库软件的安装，以及数据内部文件的设置，如根据用户输入的全局数据库名，以及用户密码建立相关的控制文件和参数文件等，同时部署数据库软件到安装主目录，但进度条显示 100%时，自动进入如图 1-10 所示的“配置助手”对话框，此时自动安装数据库网络助手和数据库配置助手，即 Oracle Net Configuration Assistant 和 Oracle Database Configuration Assistant。



图 1-9　数据库的概要文件列表



图 1-10　安装数据库软件

08 当数据库配置助手安装完毕后，安装程序自动进入如图 1-11 所示的对话框，此时使用配置助手来复制数据库文件，创建并启动数据库实例的同时创建了数据库。

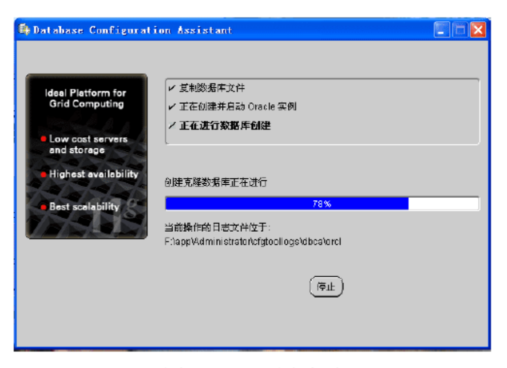


图 1-11　配置助手

09 当图 1-11 的数据库创建完毕后，自动弹出如图 1-12 所示的对话框，提示创建数据库完成，并给出安装数据库过程中设置的参数信息，如全局数据库名等，在安装 Oracle 11g 数据库时，除了在图 1-3 中设置过口令的几个用户外，其他所有用户都被锁定，需要在数据库安装成功后解锁这些用户，此时也可使用如图 1-12 所示的口令管理来解锁需要的用户，并且修改这些用户的密码，显然 Oracle 这样的考虑增强了数据库软件的安全性。单击图 1-12 中的“确定”按钮，弹出如图 1-13 所示的对话框，提示“安装结束”。

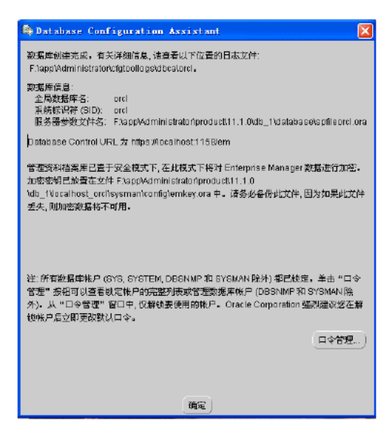


图 1-12　安装结束



图 1-13　Oracle 的企业管理器

10 单击图 1-13 中的“退出”按钮退出对话框，完成整个 Oracle 11g 数据库软件的安装。

1.1.3　SQL*Plus 工具和 SCOTT 用户

既然成功安装了数据库，就需要维护和管理它，Oracle 提供了一个工具 SQL*Plus 来完成数据库的管理和维护任务，虽然 Oracle 较高版本的数据库软件中提供了大量的图形化管理手段，但是学会使用 SQL*Plus 工具仍然是 DBA 的一项重要基本功，因为有时图形化工具或许不能使用，而 SQL*Plus 却能随时运行。

在 Oracle 11g 中打开 SQL*Plus 工具，如图 1-14 所示。

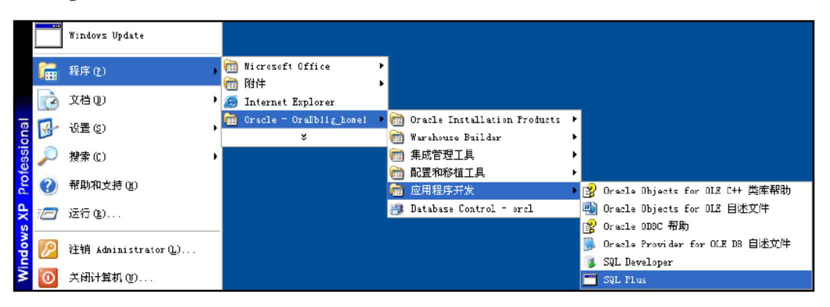


图 1-14　打开 SQL*Plus

单击“SQL Plus”选项后，弹出如图 1-15 所示的对话框。此时提示“请输入用户名”，输入 SCOTT，此时会继续提示“输入口令”，如图 1-16 所示，在口令中输入 TIGER，在输入用户名和口令后，按“回车”键，执行结果如图 1-17 所示。

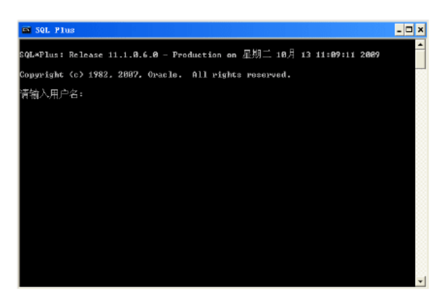


图 1-15　SQL*Plus 对话框

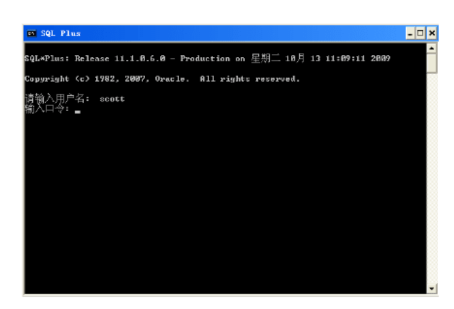


图 1-16　使用 SCOTT 用户登录数据库

在图 1-17 中提示错误（ERROR），错误信息是“the account is locked”，显然数据库告知用户 SCOTT 被锁定了。只有解锁后才可以使用该用户登录数据库。

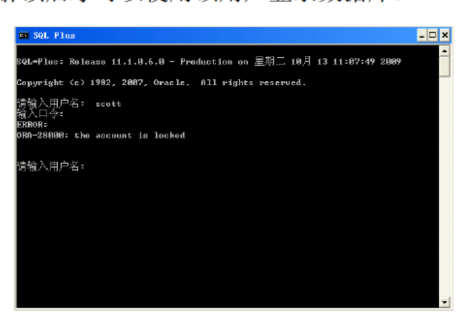


图 1-17　无法使用锁定的用户登录数据库

在解释图 1-3 时讲过，除了那 4 个用户外，其他所有用户都已经锁定了，所以可使用 SYSTEM 用户登录，密码为 oracle，再次输入用户名和密码，如图 1-18 所示，数据连接成功，进入“SQL>”状态。此时就可以使用 SQL*Plus 完成数据库的维护工作了，因为在以后的学习中，需要使用 SCOTT 用户来学习使用各种 SQL 语句，所以先解锁 SCOTT 用户，在解锁 SCOTT 用户时，需使用数据库维护指令 alter user scott identified by tiger account unlock，如图 1-19 所示。

图 1-18　用 SYSTEM 用户登录数据库

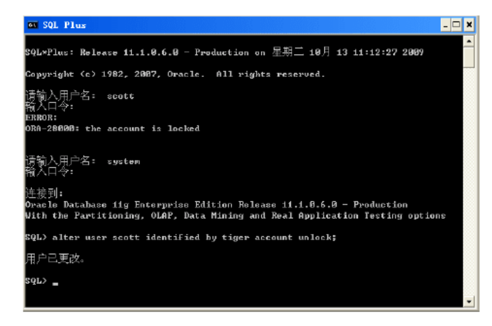


图 1-19　解锁 SCOTT 用户

如图 1-19 所示，输入的指令执行成功，因为提示“用户已更改”，此时就可以使用 SCOTT 用户了。



在安装数据库时，用户 SCOTT 是默认安装的，而且密码默认为 TIGER，其实 SCOTT 是 Oracle 数据库公司早期的一个程序员，而 TIGER 则是他的一只宠物猫的名字，或许是为了记住这位早期优秀的数据库程序员而一直保留了这个用户。

通过在 SQL*Plus 中输入各种类型的 SQL 语句或者执行某些 SQL 脚本（由多行 SQL 语句组成）来维护和管理数据库。如果读者在使用 SCOTT 用户学习时，出现一些意外情况，如删除了数据无法恢复，或者该用户被他人删除等，可以使用一个脚本来恢复。该脚本文件位于 $ORACLE_HOME\RDBMS\ADMIN 下，脚本文件名为 scott.sql。$ORACLE_HOME 为本机安装 Oracle 数据库的主目录，即 F:\app\Administrator\product\11.1.0\db_1，所以执行脚本文件的指令如下所示。

```
SQL> @ F:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN\scott.sql
```

下面再提供一种启动 SQL*Plus 的方法，并使用解锁的 SCOTT 用户登录。具体步骤如下：

01 打开一个 DOS 窗口，如图 1-20 所示。

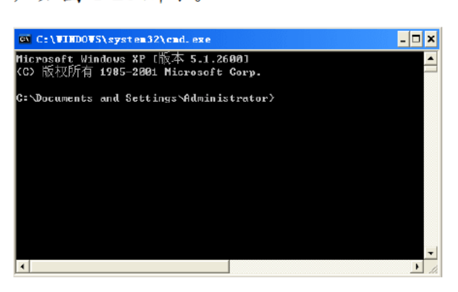
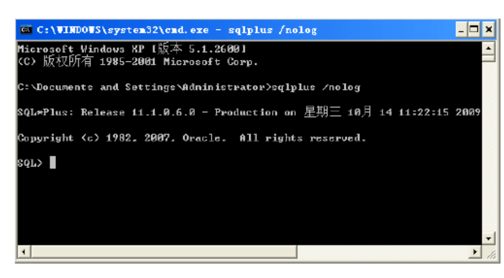


图 1-20　打开 DOS 窗口

02 在图 1-20 中输入“sqlplus /nolog”来启动 SQL*Plus 工具，如图 1-21 所示。

03 SCOTT 用户通过 CONNECT 指令登录数据库，如图 1-22 所示。



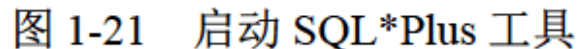


图 1-21　启动 SQL*Plus 工具

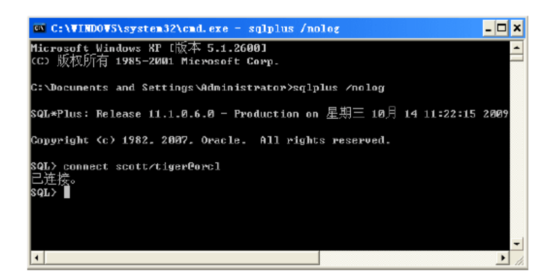


图 1-22　SCOTT 用户登录数据库

1.2　卸载 Oracle 数据库

卸载 Oracle 数据库比较简单，本节使用 Universal Installer 来卸载 Oracle 数据库，具体步骤如下。

01 启动 Universal Installer 工具，如图 1-23 所示。

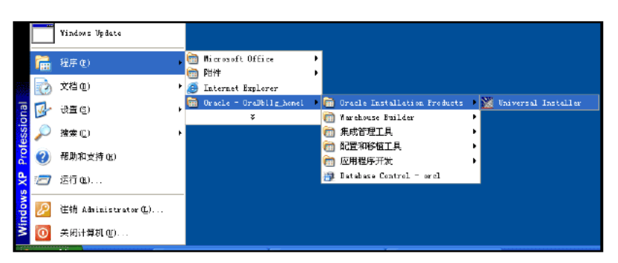


图 1-23　启动 Universal Installer 工具

02 弹出如图 1-24 所示的“正在启动 Oracle Universal Installer”卸载 Oracle 数据库的界面。

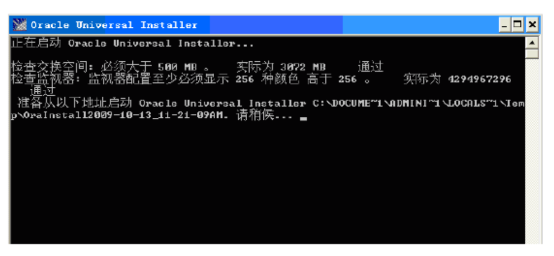


图 1-24　启动 Oracle Universal Installer

03 在图 1-24 中的启动过程完毕后，自动弹出如图 1-25 所示的卸载数据库的对话框，单击图 1-25 中的“下一步”按钮，弹出“产品清单”对话框，如图 1-26 所示。

04 在如图 1-26 所示的对话框中，选择要卸载的 Oracle 产品，因为这里要卸载整个数据库，所以选择全部数据库组件，单击“删除”按钮，弹出如图 1-27 所示对话框。

05 为了安全起见，系统会再次提示用户确认在图 1-26 中选择的要删除的数据库组件，单击“是（Y）”按钮，弹出警告对话框，如图 1-28 所示，继续执行删除操作，再次单击“是（Y）”按钮，开始执行删除数据库操作，如图 1-29 所示。当图 1-29 的删除操作完成后，会自动弹出如图 1-30 所示对话框，说明已经没有了“Oracle 产品”，且已删除所有数据库组件。

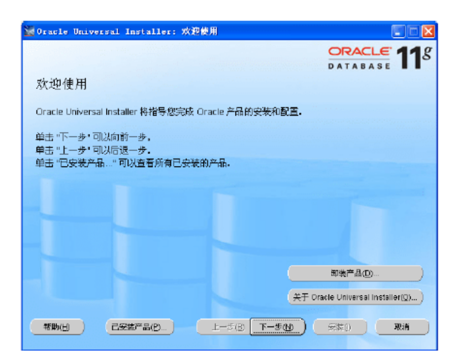


图 1-25　卸载 Oracle 数据库

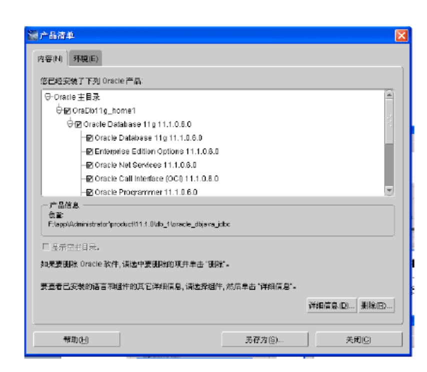


图 1-26　选择卸载组件

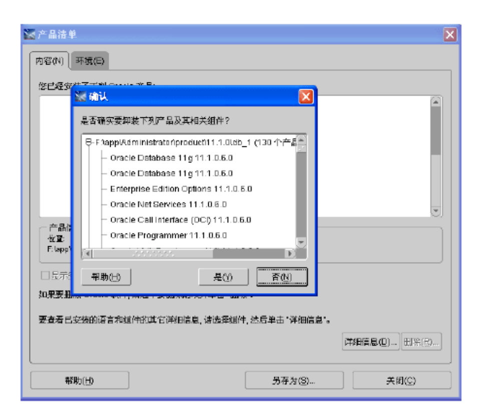


图 1-27　确认删除的数据库组件

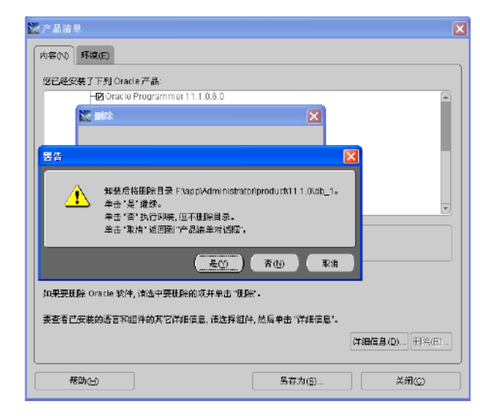


图 1-28　删除前的警告提示

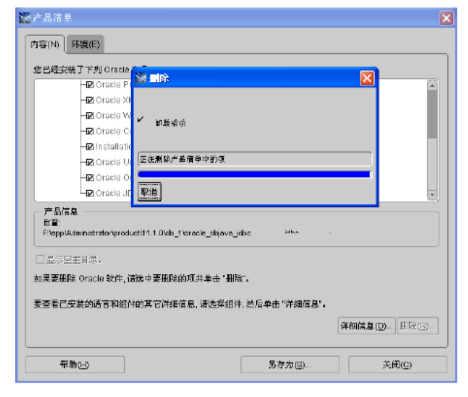


图 1-29　删除所选组件



图 1-30　卸载完毕

06 单击图 1-27 中的“关闭”按钮，退出数据库卸载程序，此时会返回到 Oracle Universal Installer 对话框，如图 1-31 所示。单击“取消”按钮，退出 Oracle Universal Installer。

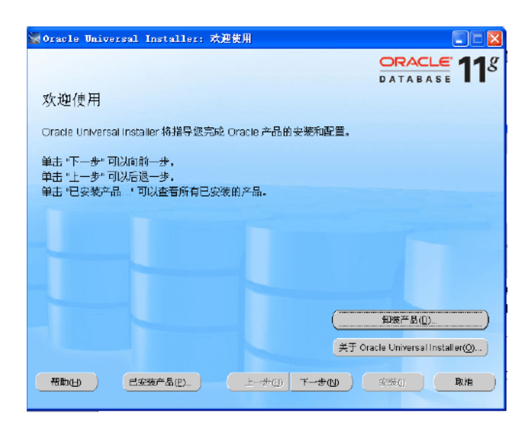


图 1-31　返回到 Oracle Universal Installer 界面



在执行上述卸载数据库过程后，需要在 C 盘的 programs 文件夹中手工删除文件夹 oracle，然后删除 Oracle 数据库软件安装目录的 app 文件夹，但是此时会提示无法删除，提示信息如图 1-32 所示。解决方法是重新启动计算机，再将 Oracle 数据库软件安装目录的 app 文件夹删除。

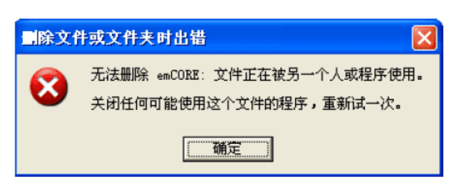


图 1-32　提示信息

1.3　本章小结

本章重点讲解了如何安装数据库软件，在安装相应版本的数据库之前需要理解该软件对于计算机硬件的需求，然后使用相应版本的数据库安装软件启动安装，整个安装过程比较简单，只须设置几个参数，一般情况下只须单击“下一步”按钮即可完成安装。

SQL*Plus 是 Oracle 中非常重要的、也是很“古老”的数据库维护和管理工具，对于 DBA 而言，需要很好地掌握这个工具，它提供了维护和管理数据库更大的灵活性以及自由度，并且相对于图像化管理工具更具有稳定可靠的特点。而对于 SCOTT 用户，读者需要知道它是 Oracle 数据库默认的一个用户，默认密码为 TIGER，在开始安装数据库时，如果没有解锁，则该用户被锁定，需要重新登录数据库解锁该数据库，使用 SCOTT 用户的一些表作为学习 Oracle 数据库的表数据对象，而且该用户损坏可以很方便地使用脚本文件恢复。最后讲解了如何使用 Universal Installer 卸载 Oracle 数据库，这里只要按照对话框的说明进行操作，单击“下一步”按钮即可顺利删除数据库软件，但是它不会删除 Oracle 一些安装目录下的文件，所以需要读者手动删除。

第 2 章

Oracle数据库体系结构

本章将讲解 Oracle 体系结构，这是全书非常重要的一章，学习 Oracle 数据库，一开始从宏观上把握它的物理组成、文件组成和各种管理进程，对于进一步的学习起到很好地指导作用，不会使读者觉得在学习某一部分知识时，只见树木不见森林。相反如果读者基本掌握或者理解了 Oracle 数据库体系结构的知识，就可以从更高的角度看待以后学习的内容。希望读者在学习本章时要细细体会、揣摩。本章将紧紧围绕数据库体系结构图详细介绍每一个数据库系统结构组件。

2.1 Oracle 数据库的体系结构概述

体系结构是对一个系统的框架描述，是设计一个系统的宏观工作。如同建一座大楼需要设计图纸一样，根据建筑框架图的要求，“严格”施工就可以建造一座功能完善、质量可靠的建筑。即使大楼建好以后，依然可依据设计图纸来找到几乎每一个功能部件。

数据库系统结构设计了整个数据库系统的组成和各部分组件的功能，这些组件各尽其职，相互协调完成数据库的管理和数据库维护工作。

为了满足数据库需求，Oracle 设计了如图 2-1 所示的体系结构，该体系结构包括：实例（Instance）、数据库文件、用户进程（User process）和服务器进程（Server Process）以及其他文件，如参数文件（Parameter File）、密码文件（Password File）和归档日志文件（Archived Log File）等。

其中，数据库实例包括 SGA（系统全局区）和一系列后台管理、监视进程，数据库包括三种文件：数据文件（Data Files）、控制文件（Control Files）和重做日志文件（Redo Log Files）。数据库实例和数据库是 Oracle 数据库体系结构的核心部分，DBA 很重要的工作就是维护实例和数据库本身的正常工作。

本节将依次介绍数据库服务器（包括实例和数据库）和实例、数据库的物理结构、密码文件和参数文件。

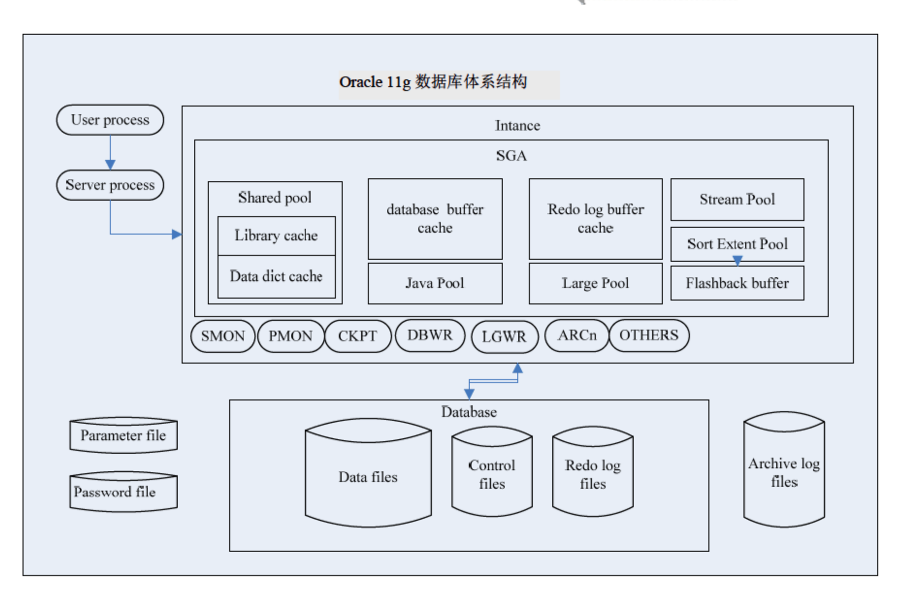


图 2-1　Oracle 数据库体系结构

2.1.1　服务器和实例

Oracle 服务器和实例是非常重要的两个概念，这里的服务器不仅仅是物理概念，还包括系统进程，而实例则是 DBA 经常维护的对象。

1. Oracle 实例（Instance）

Oracle 实例由一些内存区和后台进程组成，如图 2-2 所示，这些内存区包括数据库高速缓存、重做日志缓存、共享池、流池以及其他可选内存区（如 Java 池），这些池也称为数据库的内存结构。后台进程包括系统监控进程（SMON）、进程监控（PMON）、数据库写进程（DBWR）、日志写进程（LGWR）、检验点进程（CKPT）、其他进程（SMON，如归档进程、RECo 进程等），这些数据库系统进程相互协作完成数据管理任务。

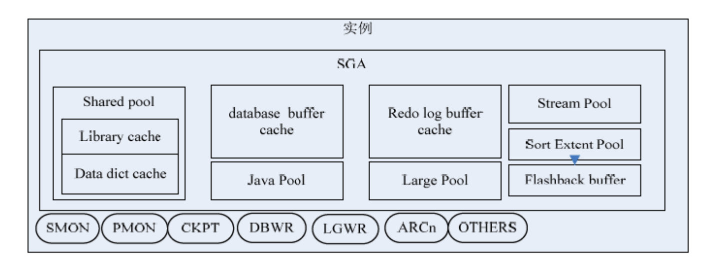


图 2-2　Oracle 实例（Instance）组成图

要访问数据库必须先启动实例，实例启动时先分配内存区，然后再启动后台进程，后台进程执行数据的输入、输出以及监控其他 Oracle 进程。在数据库启动过程中有 5 个进程是必须启动的，它们是系统监控进程（SMON）、进程监控（PMON）、数据库写进程（DBWR）、日志写进程（LGWR）、

检验点进程（CKPT），否则实例无法创建。数据库启动过程可以在告警日志（alertSID.ora）中看到详细的流程。

> 注意　在实践中，为了方便，会通过数据库工具实现数据库在计算机重启时自动启动，如果用户安装了其他占用大量内存的应用软件，可能会造成数据库启动失败，此时往往是因为内存不足，操作系统无法为 Oracle 分配 SGA，必须的 5 个进程也无法启动。

2. Oracle 服务器（Server）

Oracle 服务器由数据库实例和数据库文件组成，即经常说的数据库管理系统（DBMS）。数据库服务器的组成如图 2-3 所示。

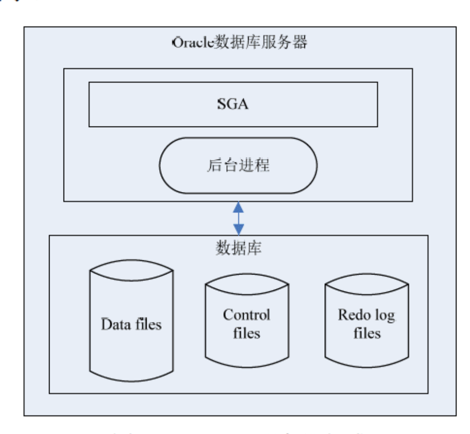


图 2-3　Oracle 服务器组成

数据库服务器除了维护实例和数据库文件外，还在用户建立与服务器的连接时启动服务器进程并分配 PGA（程序全局区）。

2.1.2　物理结构

数据库是运行在操作系统之上的，数据库的最终目的就是存储和获取相关的数据，这些数据实际上存储在操作系统文件中，这些操作系统文件组成 Oracle 数据库物理结构。

Oracle 数据库的物理结构就是指数据库中的一系列操作系统文件，Oracle 数据库由三类文件组成。

- 数据文件（Data Files）：数据文件包含数据库中的实际数据，是数据库操作中数据的最终存储位置。
- 控制文件（Control File）：包含维护数据库和验证数据库完整性的信息，它是二进制文件。
- 重做日志文件（Redo File）：重做日志文件包含数据库发生变化的记录，在发生故障时用于数据恢复。

2.1.3　参数文件和密码文件

虽然参数文件和密码文件不是 Oracle 的数据库文件，但却是 Oracle 数据库不可缺少的两个文件。

- 参数文件（Parameter File）：参数文件中定义了数据库实例的特性。在参数文件中包含为SGA 中内存结构分配空间的参数，如分配数据库高速缓冲区的大小等，参数文件是正文文件，可以使用操作系统文本编辑器查看，如在 Windows 操作系统中使用记事本工具。
- 密码文件（Password File）：密码文件授予用户启动和关闭数据库实例，在刚安装数据库时，Oracle 的默认用户名和密码就存储在密码文件中，Oracle 可以藉此判断用户的操作权限。

2.2　Oracle 数据库的连接与会话

连接与会话是 Oracle 数据库中容易混淆的两个概念，本节将讲解它们的区别，并给出实际的例子，以帮助读者更好地理解它们的概念。

2.2.1　连接

连接是指用户进程与数据库服务器之间的通信途径，一个连接可以有多个对话。Oracle 提供了三种数据库连接方式，以满足用户不同的连接需求，三种连接方式如下。

- 基于主机的方式（Host-Based）：此方式中，服务器和客户端运行在同一台计算机上，用户可以直接连接数据库服务器。
- 基于客户机-服务器的方式（Client-Server）：该方式中数据库服务器和客户端运行在不同的计算机上，客户通过网络连接数据库服务器。在 DBA 的日常维护中，会经常使用这种方式访问数据库，从而实现数据库的远程维护。
- 用户-应用服务器-数据库服务器方式（Client-Application Server-Server）：这种方式称为三层访问模式，用户首先访问应用服务器，然后由应用服务器连接数据库服务器，应用服务器就如同一个中介，用于完成客户和数据库的交互。在很多应用系统中，客户的应用程序往往通过三层方式访问数据库，如应用服务器为 IIS 或 Apache 服务器等。

2.2.2　会话

会话是指一个明确的数据库连接。一旦用户采用一种连接方式，这样的连接就称为一个会话。如用户通过某种工具（如 SQL*Plus）在专有连接的情况下访问数据库，在输入的用户名和密码经过服务器验证后，服务器就会自动创建一个与该用户进程对应的服务器进程，二者是一对一的关系，这里服务器进程就像用户进程的代理，代替用户进程向数据库服务器发出各种请求，并把从数据库服务器获得的数据返回给用户进程。但用户退出或发现异常时（操作系统重启）会话结束。



刚才指出“专有连接”的概念，专有连接是一种连接类型，指用户和服务器进程之间是一对一的关系。而在共享服务器配置的情况下，多个用户进程可以同时共享服务器进程，此时就不是专有连接，而是多对一的关系。

一个用户可以并发地建立多个会话，实例 2-1 就是用户 SYS 同时在专有连接的情况下建立两个会话的例子。

【实例 2-1】用户 SYS 通过专有连接建立两个会话。

```
SQL>SELECT serial#,username,status,server,process,program,logon_time
  2* FROM v$session
SERIAL# USERNAME STATUS  SERVER   PROCESS   PROGRAM   LOGON_TIME
----------- ---------------- ---------- ---------------- ---------------- --
     1          ACTIVE   DEDICATED 2172     ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 528      ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 2188     ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 408      ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 1424     ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 1244     ORACLE.EXE     17-5 月 -09
     1          ACTIVE   DEDICATED 2264     ORACLE.EXE     17-5 月 -09
     3 SYS      ACTIVE   DEDICATED 540:1644   sqlplus.exe    17-5 月-09
     7 SYS      ACTIVE   DEDICATED 1296:1848  sqlplusw.exe   17-5 月-09

已选择 9 行。
```

在实例 2-1 的输出中同时使用两个 SQL*Plus 工具连接数据库，并且使用同一个用户 SYS，最后两行显示有两个活跃（ACTIVE）的会话，一个会话是使用 sqlplus.exe 程序建立的，另一个是使用 sqlplusw.exe 程序建立的。



在上述查询中，只是演示一个用户可以建立多个连接，使用相同或不同的工具登录。至于 v$session（数据字典视图），读者可暂且把它看成是一张表，表中存储了当前会话的信息，如属性 USERNAME 是用户登录名，属性 PROGRAM 是用户登录工具（一个用户进程）。

图 2-4 清晰地说明了连接与会话之间的区别和联系。

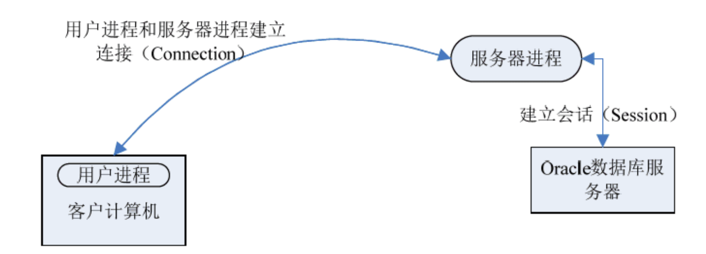


图 2-4　连接与会话示意图



一个连接可以对应多个对话，连接仅仅是一种通信途径，如通过 Socket 建立通信，但是一个用户可以启动多个进程通过一个连接建立多个对话，这里服务器进程就像是用户进程的代理一样，与服务器交互完成数据的各种操作。

2.3　Oracle 数据库的内存结构

Oracle 的内存结构由两大部分组成，即 SGA 和 PGA。PGA 称为程序全局区，程序全局区不是实例的一部分，当服务器进程启动时才分配 PGA。而 SGA 称为系统全局区，它是数据库实例的一部分，当数据库实例启动时，会首先分配系统全局区。

2.3.1　SGA

在系统全局区中包含几个重要的内存区，即数据库高速缓存（Database Buffer Cache）、重做日志缓存（Redo Log Buffer Cache）、共享池（Shared Pool）、大池（Large Pool）和 Java 池（Java Pool）。

1．共享池

Oracle 引入共享池的目的就是共享 SQL 或 PL/SQL 代码，即把解析得到的 SQL 代码的结果在这里缓存，其中 PL/SQL 代码不仅在这里缓存，同时还在这里共享。共享池由两部分组成，即库高速缓存和数据字典高速缓存，如图 2-5 所示。

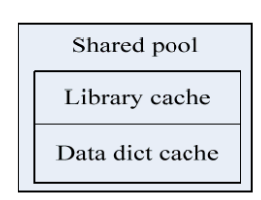


图 2-5　共享池的组成

（1）库高速缓存

库高速缓存存储了最近使用过的 SQL 和 PL/SQL 语句。当然它的容量是有限的，Oracle 采用一种 LRU（Least Recently Used）算法管理库高速缓存，算法的基本思想是把一段时间内没有被使用过的语句清除，一旦缓冲区填满，算法就把最近很少使用的执行计划和解析树从库高速缓存中清除。显然库高速缓存设置得越大，就可以共享更多的 SQL 或 PL/SQL 代码，但是 Oracle 并没有设计直接设置库高速缓存的指令，只能通过设置共享池的大小间接地更改，而共享池是 SGA 的一部分，所以共享池不能超过 SGA 的大小。

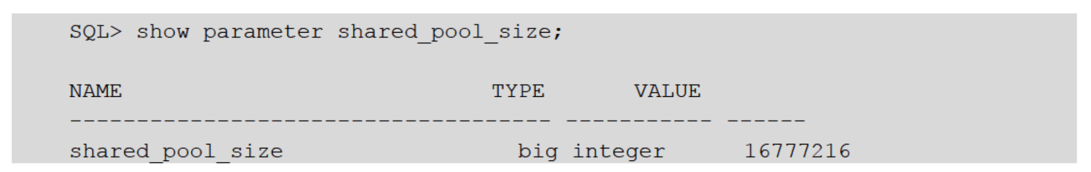
【实例 2-2】设置共享池的大小。

```
SQL> alter system set shared_pool_size = 16M;

系统已更改。
```

通过实例 2-3 验证修改结果。

【实例 2-3】查看共享池的大小。



```
SQL> show parameter shared_pool_size;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------
shared_pool_size                     big integer 16777216
```

（2）数据字典高速缓存

顾名思义，该缓存区是与数据字典相关的一段缓冲区。在数据字典高速缓冲区中存储了数据

文件、表、索引、列、用户、权限信息和其他一些数据库对象的定义。在 SQL 语句的解析阶段，数据库服务器需要这些信息来解析用户名和用户的访问权限。如果 Oracle 缓存了这些信息，无疑提高了查询的响应时间。

数据字典缓存也称为字典缓存或者行缓存，无论称呼如何，读者只需要理解它的作用，就是把相关的数据字典信息放入缓存以提高查询的响应时间。

同样数据字典高速缓存的大小取决于共享池尺寸的大小。如果设置的太小，但查询需要数据字典信息时，Oracle 将不断地访问数据字典表来获得所需的信息，由于数据字典也是存储在磁盘上的一类数据文件，频繁地磁盘 I/O 无疑降低了数据库的查询速度。如果需要设置字典高速缓存的大小，需要通过设置 shared_pool_size 间接实现。

2. 数据库高速缓存

数据库高速缓存中存储了最近从数据文件读入的数据块信息或用户更改后需要写回数据库的数据信息，此时这些没有提交给数据库的更改后的数据称为脏数据。当用户执行查询语句，如 select * from dept 时，如果用户查询的数据块在数据库高速缓存中，Oracle 就不必从磁盘读取，而是直接从数据库高速缓存中读取，显然物理读取的速度比从内存读取的速度慢很多，这些缓存的数据由 LRU 算法管理。可见 Oracle 设计的各种缓存目的基本相同，就是提高查询速度，减少用户查询的响应时间。Oracle 使用 LRU 算法管理库缓冲区，把最近没被使用的数据库从库高速缓存中删除，为其他的查询数据块保留空间。

Oracle 使用参数 DB_BLOCK_SIZE 和 DB_BLOCK_BUFFERS 设置库高速缓存的大小，DB_BLOCK_SIZE 是 Oracle 数据块的大小，而 DB_BLOCK_BUFFERS 是数据库的个数，二者的乘积就是库高速缓存的大小。

【实例 2-4】查看数据库块大小。

```
SQL> show parameter db block size;

NAME                                TYPE        VALUE
---------------------------------------------------------- ----------
db_block_size                               integer      8192
```



用这种方式设置数据库高速缓存的大小需要重启数据库才能生效，db_block_size 的值是 8192 字节（即 8K 字节）。

在 Oracle 9i 及以上版本中提供了一个 DB_CACHE_SIZE 参数来设置 Oracle 数据库高速缓存区的大小，该参数可以动态更改，之后可以通过查询指令查看更改后的参数。

【实例 2-5】查询数据库高速缓存的大小。

```
SQL> show parameter db_cache_size;

NAME                                TYPE        VALUE
-------------------------------------------------------- ----------------
db_cache_size                              big integer     0
```

因为在 Oracle 11g 中，SGA 为数据库服务器自动管理，所以该参数值为 0，当然在运行 Oracle 11g 数据库时，数据库高速缓存是一定已分配好的，所以使用 show sga 指令可查看数据库高速缓冲区分配的内存大小。

【实例 2-6】查询数据库高速缓存的大小。

```
SQL> show sga;

Total System Global Area  535662592 bytes
Fixed Size                  1334380 bytes
Variable Size             260047764 bytes
Database Buffers          268435456 bytes
Redo Buffers                5844992 bytes
```



上述指令用于查询 SGA 的分配情况，其中 Database Buffers 为数据库缓存区的大小。我们更改的值为 32M，而显示的值为 33554432 字节，二者一致，说明修改成功（32M=32*1024*1024bytes=33554432 bytes）。

虽然在 Oracle 11g 中数据库高速缓存的大小为自动管理，但是用户可以设置该数据库组件的大小。

【实例 2-7】动态设置数据库高速缓冲区大小。

```
SQL> alter system set db_cache_size = 200M;

系统已更改。
```

在 Oracle 中引入了 Buffer Cache Advisory Parameter 参数，其目的是让 Oracle 对于数据库缓冲区的内存分配提供一些建议，下面介绍缓冲区顾问参数（Buffer Cache Advisory Parameter）的作用，缓冲区顾问参数用于启动或关闭统计信息，这些信息用于预测不同缓冲区的大小导致的不同行为特性。对于 DBA 而言，可以参考这些统计信息，基于当前的数据库工作负载设置优化的数据库高速缓存。

缓存顾问通过初始化参数 DB_CACHE_ADVICE 启动或关闭顾问功能，该参数有三个状态。

- OFF：关闭缓存顾问，不分配缓存顾问的工作内存。
- ON：打开缓存顾问，分配工作内存。
- READY：打开缓存顾问，但不分配缓存顾问的工作内存。

【实例 2-8】当前缓存顾问的状态。

```
NAME                                    TYPE        VALUE
-------------------------------------------------------- -----------------
db_cache_advice                         string      ON
```

从上述输出中可以看出，参数 db_cache_advice 的值为 ON，所以默认是打开缓存顾问的。实例 2-9 演示了如何设置缓存顾问为关闭状态，是通过设置参数 db_cache_advice 实现的。

【实例 2-9】关闭数据库高速缓存顾问。

```
SQL> alter system set db_cache_advice = off;
```

```
系统已更改。
```

在更改了缓存顾问状态后，通过实例 2-10 查看当前的缓存顾问状态，以验证更改结果。

【实例 2-10】查看数据库高速缓存顾问状态。

```
SQL> show parameter db_cache_advice;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---
db_cache_advice                        string      OFF
```

当然我们的目的是使用缓存顾问，所以需要再次将参数 db_cache_advice 的值设置为 ON，如下所示：

```
SQL> alter system set db_cache_advice = on;

系统已更改。
```

在设置顾问缓存为开启状态后，Oracle 开始统计与设置数据库缓存相关的建议信息，可以通过动态性能视图 v$DB_CACHE_ADVICE 查看缓冲区的建议信息，如实例 2-11 所示。

【实例 2-11】查看与设置数据库高速缓冲区相关的信息。

```
SQL> col id for 99
SQL> SELECT id, name, block_size, size_for_estimate, buffers_for_estimate
  2  FROM v$db cache advice;

ID NAME     BLOCK_SIZE ADV SIZE_FOR_ESTIMATE BUFFERS_FOR_ESTIMATE
-- ---------------- -------------------- ---------- --- ------------------ ----
3 DEFAULT          4096  ON              3.0703                  786
3 DEFAULT          4096  ON              6.1406                  1572
3 DEFAULT          4096  ON              9.2109                  2358
3 DEFAULT          4096  ON             12.2813                  3144
3 DEFAULT          4096  ON             15.3516                  3930
3 DEFAULT          4096  ON             18.4219                  4716
3 DEFAULT          4096  ON             21.4922                  5502
3 DEFAULT          4096  ON             24.5625                  6288
3 DEFAULT          4096  ON             27.6328                  7074
3 DEFAULT          4096  ON             30.7031                  7860
3 DEFAULT          4096  ON             33.7734                  8646

ID NAME      BLOCK_SIZE ADV SIZE_FOR_ESTIMATE BUFFERS_FOR_ESTIMATE
-- ---------------- -------------------- ---------- --- --------------------
3 DEFAULT          4096  ON             36.8438                   9432
3 DEFAULT          4096  ON             39.9141                   10218
3 DEFAULT          4096  ON             42.9844                   11004
3 DEFAULT          4096  ON             46.0547                   11790
3 DEFAULT          4096  ON              49.125                   12576
3 DEFAULT          4096  ON             52.1953                   13362
3 DEFAULT          4096  ON             55.2656                   14148
```

```
3 DEFAULT          4096 ON          58.3359                14934
3 DEFAULT          4096 ON          61.4063                15720

已选择 20 行。
```

3. 重做日志缓存

当用户执行了如 INSERT、UPDATE、DELETE、CREATE、ALTER 和 DROP 操作后，数据发生了变化，这些变化了的数据在写入数据库高速缓存之前会先写入重做日志缓冲区，同时变化之前的数据也放入重做日志高速缓存，这样在数据恢复时，Oracle 就知道哪些需要前滚、哪些需要后滚。

重做日志缓冲区的大小是可动态调节的，即在数据库运行期间修改这块内存的大小，Oracle 提供了一个初始化参数 LOG_BUFFER，在数据库实例启动时就分配好重做日志缓冲区的尺寸。实例 2-12 演示了如何查看重做日志缓存区。

【实例 2-12】查看重做日志缓存区。

```
SQL> show parameter log_buffer;

NAME                                TYPE        VALUE
---------------------------------------------------------- -
log_buffer                                integer       5653504
```



说 明

重做日志缓存区参数 log_buffer 是静态参数，不能动态修改，如果尝试修改会提示如下错误：

```
SQL> alter system set log_buffer = 1M;
alter system set log_buffer = 1M
                 *
ERROR 位于第 1 行:
ORA-02095: 无法修改指定的初始化参数
```

4. 大池和 Java 池

大池是 SGA 的一段可选内存区，只在共享服务器环境中配置大池（Large Pool）。在共享服务器环境下，Oracle 在共享池中分配额外的空间用于存储用户进程和服务器进程之间的会话信息，但是用户进程区域 UGA（可理解为 PGA 在共享服务器中的另一个称呼）的大部分将在大池中分配，这样就减轻了共享池的负担。在大规模输入、输出及备份过程中也需要大池作为缓存空间。

Oracle 提高了参数 large_pool_size 设置大池的尺寸。

【实例 2-13】查看大池大小。

```
SQL> show parameter large pool size

NAME                                TYPE        VALUE
----------------------------------------------------------- ---
large_pool_size                          big integer    52M
```

参数 large_pool_size 是动态参数，可以通过 alter system 指令修改该参数的值，语句格式为：

"SQL>alter system set large_pool_size = 48M;"。

Java 池也是可选的一段内存区，但是在安装了 Java 或者使用 Java 程序时必须设置 Java 池，它用于编译 Java 语言编写的指令。Java 语言与 PL/SQL 语言在数据库中有相同的存储方式。Oracle 提供了参数 java_pool_size 来设置 Java 池的大小。

【实例 2-14】查看 Java 池的大小。

```
SQL> show parameter java_pool_size;

NAME                                 TYPE        VALUE
------------------------------------ ----------- -------
java_pool_size                          big integer        0
```



在 Oracle 11g 中，Java 池的大小由数据库服务器在 SGA 中自动分配，当然用户也可以使用 alter system 指令修改该参数的值，在实例 2-14 中参数 java_pool_size 的值为 0 说明该参数为自动管理。

2.3.2 PGA

进程全局区（PGA）是服务器进程专用的一块内存，系统中的其他进程是无法访问这块内存的。PGA 独立于 SGA，PGA 不会在 SGA 中出现，它是由操作系统在本地分配的。

PGA 中存储了服务器进程或单独的后台进程的数据信息和控制信息。它随着服务器进程的创建而被分配内存，随着进程的终止而释放内存。PGA 与 SGA 不同，它不是一个共享区域，而是服务器进程专有的区域。在专有服务器（与共享服务器相对的概念）配置中包括如下的组件。

- 排序区：对某些的 SQL 语句执行结果进行排序。
- 会话信息：包含本次会话的用户权限和性能统计信息。
- 游标状态：表明当前会话执行的 SQL 语句的处理阶段。
- 堆栈区：包含其他的会话变量。



在共享服务器配置中，多个用户进程共享一个服务器进程，上述的一些内存区可能在 SGA 中分配。如果创建了大池（Large Pool），这些内存结构就存储在大池中，否则它们存储在共享池中。

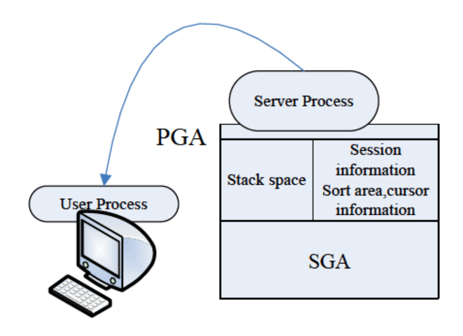
图 2-6 和图 2-7 分别是专有服务器模式和共享服务器模式下的 PGA 结构图。



在共享服务器结构中，会话信息是存储在 SGA 中的，两种模式下堆栈区（Stack space）都是存储在 PGA 中。

从 Oracle 9i 开始，Oracle 提供了两种办法来管理 PGA，即手动 PGA 管理和自动 PGA 管理。采用手动管理时，必须告诉 Oracle 一个特定的进程需要的排序区，以及允许使用多少内存。而在自动 PGA

管理中，则要求高速 Oracle 在系统范围内可以为 PGA 中的特定功能，如排序区分配多少内存。



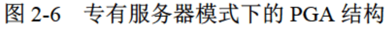


图 2-6　专有服务器模式下的 PGA 结构

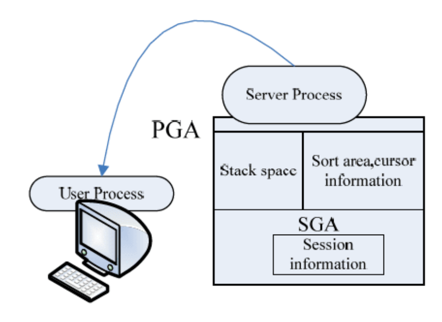


图 2-7　共享服务器模式下的 PGA 结构

【实例 2-15】查询 PGA 中排序区的大小。

```
SQL> show parameter sort_area_size;

NAME                                TYPE        VALUE
--------------------------------------------------------- ------
sort_area_size                                  integer       65536
```

在服务器进程最初查询时，会使用 512K 内存实现数据排序，在 Oracle 将排序数据处理完之前，数据排序区的大小就由参数 SORT_AREA_SIZE 决定。



在 Oracle 10g 和 Oracle 11g 中可以实现共享服务器连接时 PGA 的自动管理，而在 Oracle 9i 中，使用共享服务器连接时，只能使用手动 PGA 管理。

2.3.3　如何获得内存缓冲区的信息

SGA 是 Oracle 中所有进程共享的一段内存区，其中共享了数据库信息，如数据库高速缓冲区中的数据、共享池中的库高速缓存中的 SQL 语句等。了解这些内存缓冲区的大小有助于理解 Oracle 的内存分配情况。

【实例 2-16】查看 SGA 中内存的分配情况。

```
SQL> show sga;

Total System Global Area  535662592 bytes
Fixed Size                  1334380 bytes
Variable Size             260047764 bytes
Database Buffers          268435456 bytes
Redo Buffers                5844992 bytes
```

在上述输出中可以看到 SGA、Database Buffers 和 Redo Buffers 的尺寸，前面已经讲解了这些内存组件的作用。读者或许注意到了 Fixed Size 和 Variable Size 两个参数，它们和以下两个内存区有关。

● 固定 SGA（和 Fixed Size 相关）：在固定 SGA 中，存储一组指向 SGA 中其他组件的变量。

用户无法控制它的大小，因平台不同而有差异。但通常固定 SGA 区很小。Oracle 使用这个内存区来寻找其他 SGA 区，可以理解为数据库的自举区。

- 和 Variable Size 相关的内存区：该部分内存区包括共享池、Java 池和大池，其中 Variable Size 的尺寸要高于上述三个内存结构之和，因为在 Total SGA 中除去 db_cache_size 部分也包括在 Variable Size 中。

同时读者也可以使用实例 2-17 查询当前数据库的 SGA 尺寸。

【实例 2-17】查看 SGA 的尺寸。

```
SQL> show parameter sga_max_size;

NAME                                 TYPE        VALUE
------------------------------------------------------- ---------
sga_max_size                         big integer     512M
```



说 明

在 Oracle 11g 中 sga_max_size 参数的值得到修正而使用 M 字节作为单位，更利于识别，而在 Oracle 10g 版本中参数 sga_max_size 的值使用字节为单位。

2.4 Oracle 数据库的服务器进程和用户进程

服务器进程和用户进程是用户使用数据库连接工具同数据库服务器建立连接时涉及的两个概念。

- 服务器进程：服务器进程犹如一个中介，完成用户的各种数据服务请求，而把数据库服务器返回的数据和结果发给用户端。在专有连接中，一个服务器进程对应一个用户进程，二者是一一对应的关系。当用户连接中断，则服务器程序退出。在共享连接中，一个服务器进程对应几个用户进程，此时服务器进程通过 OPI（Oracle Program Interface）与数据库服务器通信。
- 用户进程：当用户使用数据库工具如 SQL*Plus 与数据库服务器建立连接时，就启动一个用户进程，即 SQL*Plus 软件进程。

【实例 2-18】使用 SCOTT 用户连接数据库。

```
SQL> conn scott/tiger@orcl
已连接。
```

此时，用户和数据库服务器建立了连接，数据库服务器产生一个服务器进程，负责与数据库服务器的直接交互。

2.5 Oracle 数据库的后台进程

后台进程是在实例启动时，在数据库服务器端启动的管理程序，它使得数据库的内存结构和

数据库物理结构之间协调工作。从功能上考虑，在数据库物理结构、数据库内存结构和后台进程之间的关系如图 2-8 所示。

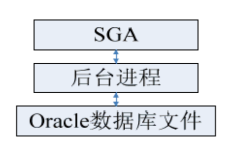


图 2-8 内存结构、后台进程和物理结构的关系图

数据库后台进程有 5 个是必须启动的，否则数据库实例无法启动成功。它们是：DBWR、LGWR、PMON、SMON 和 CKPT。本节将主要讲解这 5 个后台进程，它们也是数据库服务器中最重要的几个后台进程。

2.5.1 系统监控进程（SMON）

系统监控进程的主要作用就是数据库实例恢复。当数据库发生故障时，如操作系统重启，此时实例 SGA 中的所有没有写到磁盘的信息都将丢失。当数据库重新启动后，系统监控进程自动恢复实例。实例恢复包括如下三个步骤。

01 前滚所有没有写入数据文件而记录在重做日志文件中的数据。此时，系统监控进程读取重做日志文件，把用户更改的数据重新写入数据块。

02 打开数据库，此时或许系统监控进程的前滚操作还没有完成，Oracle 这样做的目的就是方便用户及时登录，以免前滚时间太长，影响用户的操作，这样用户就可以操作那些没有被事务恢复锁住的数据。

03 回滚未提交的事物。

除此之外，系统监控进程还执行以下空间的维护操作。

- combine、coalesces、adjacent 数据文件中的自由空间。
- 回收数据文件中的临时段。

2.5.2 进程监控进程（PMON）

进程监控负责服务器进程的管理和维护工作，在进程失败或连接异常发生时该进程负责以下一些清理工作。

- 回滚没有提交的事务。
- 释放所持有的当前的表或行锁。
- 释放进程占用的 SGA 资源。
- 监视其他 Oracle 的后台进程，在必要时重启这些后台进程。
- 向 Oracle TNS 监听器注册刚启动的实例。如果监听器在运行，就与这个监听器通信并传递，如服务名和实例的负载等参数，如果监听器没有启动，进程监控（PMON）会定期地尝试连接监听器来注册实例。

2.5.3 数据库写进程（DBWR）

在介绍高速缓冲区时，提到了脏数据的概念，脏数据就是用户更改了的但没有提交的数据库中的数据，因为数据库中的数据文件与数据库高速缓存中的数据不一致，所以称为脏数据，这种脏数据必须在特定的条件下写到数据文件中，这就是数据库写进程的作用。

数据库写进程负责把数据库高速缓冲区中的脏数据写到数据文件中。或许读者会问，为什么不立即提交脏数据呢，这样就不需要复杂的数据库写进程来管理。其实，Oracle 这样设计的思路很简单，就是减少 I/O 次数，但脏数据量达到一定程度或者某种其他条件满足时，就提交一次脏数据。因为磁盘的输入、输出会花费系统时间，使得 Oracle 系统的效率不高。

图 2-9 是数据库写进程涉及的数据库组件，通过该图可以清晰地理解数据库写进程工作中涉及的“实体”。

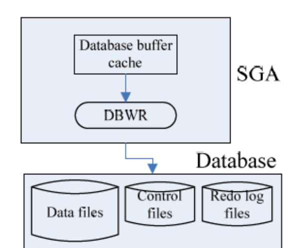
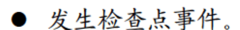


图 2-9　数据库写进程“实体”关系

当以下事件发生时，会触发数据库写进程把脏数据写到数据库的数据文件中。



- 发生检查点事件。
- 脏数据量达到了门限值。
- 数据库缓冲区没有足够的缓存为其他事务提供足够的空间。
- 表空间处于热备份状态。
- 表空间被设置为离线状态。
- 表空间被设置为只读状态。
- 删除表或者截断表。
- 超时。



注意　数据库写进程的性能显然很重要，如果它写脏数据到数据文件的速度很慢，使得大量缓冲区无法释放，就会出现一些等待事件，如 Free Buffer Waits 等。实际在 Oracle 数据库上，一个数据库实例可以启动多个数据库写进程，在多 CPU 系统中可以使用多个数据库写进程来分担单个写进程的工作负载。

2.5.4 重做日志写进程（LGWR）

重做日志写进程负责将重做日志缓冲区中的数据写到重做日志文件。此时重做日志缓冲区中的内容是恢复事务所需要的信息，如用户使用 UPDATE 语句更新了某行数据，恢复事务所需的信息就是更新前的数据和更新后的数据，这些信息用于该事务的恢复。

- 重做日志写进程在满足以下条件时会启动进程。
- 当事务提交时。
- 当重做日志缓冲区的 1/3 被占用时。

- 当重做日志缓冲区中有 1M 的数据时。
- 当数据库写进程把脏数据写到数据文件之前。
- 每 3 秒钟。

图 2-10 是重做日志写进程的工作示意图。

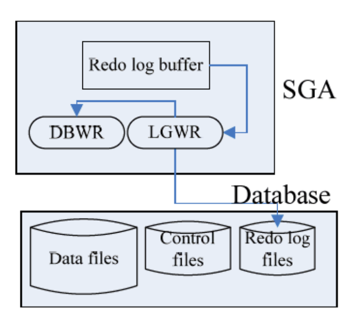


图 2-10　数据库重做日志写进程



从图 2-10 可以看出，日志写进程会通知数据库写进程将脏数据写到数据文件，但是数据库写进程不会把脏数据写到在线重做日志，也不会通知日志写进程做任何事情。

数据库写进程是离散写到不同的数据库文件上的，在执行一个更新时，数据库写进程会修改不同空间中存储的数据块和索引块，所以数据库写进程的离散写的速度很慢。而重做日志写进程是顺序写，它比离散写的效率要高，把每个事务的重做信息全部放在重做日志中。通过在数据库高速缓存中缓存脏数据块，而由重做日志写进程完成大规模顺序写，从整体上可以提高系统的性能。

2.5.5　校验点进程（CKPT）

校验点（CKPT）是一个可选进程，在系统运行中，当出现查找数据请求时，系统从数据库中找出这些数据并存入内存区，这样用户就可以对这些内存区数据进行修改等操作。当需要对被修改的数据写回数据文件时，就产生重做日志的交替写（Switch），这时就出现校验点。系统要把内存中灰数据（修改过）块中的信息写回磁盘的数据文件中，此外系统还将重做日志通知控制文件。DBA 可以改变参数文件中 CHECKPOINT_PROCESS TRUE 的值来控制（使有效或无效）该进程。

2.6　本章小结

本节首先给出 Oracle 数据库体系结构的整体框图，读者通过此结构框图可以清晰地了解各个组件。接下来把体系结构分成数据库内存结构、数据库文件结构和数据库后台进程，并逐个讲解各种结构包含的组件及其相应的功能。读者应该重点理解内存结构和后台进程，内存结构为数据库的工作分配了动态的区域，后台进程用于协调、监控数据库内存结构和物理文件之间的关系，使得数

据库系统协调地工作，从而完成数据的维护和管理任务。物理结构相对简单，包括三个文件，即数据文件、控制文件和重做日志文件。数据文件用于保存用户数据，控制文件用于保存各种启动数据库所需的信息，日志文件主要用于数据库恢复。

数据库服务器实例是很重要的概念，实例是指一组内存结构和后台进程，而数据库服务器包括实例和数据库，我们经常说的启动数据库服务器实际上是先启动数据库实例，而后挂接数据库。用户进程可以通过数据库服务器的服务器进程操作数据库。

第 3 章

数据库管理工具SQL*Plus

SQL*Plus 是 Oracle 数据库管理系统提供的一个工具软件，它提供一个人机接口，通过 SQL*Plus 管理和维护数据库，如常用的查询数据表信息、系统信息、数据文件等，它提供了一系列指令，通过这些指令可以简化用户的指令或者格式化输出信息。它还提供了编写脚本文件的功能，可以极大地提高 DBA 管理数据库的效率。SQL*Plus 作为数据库管理工具可以设置友好的环境变量，以方便 DBA 的管理和维护需求。

3.1 SQL*Plus 的常用指令

在使用 SQL*Plus 工具时，往往需要对输入的 SQL 语句进行操作，如对刚执行的指令进行修改以减少再输入全部指令的时间，如格式化列的名称，使得输出更人性化，或者对每列数据的显示宽度进行调整，使得表的输出更规范化等。本节主要讲述 SQL*Plus 提供的常用指令，并用实例演示这些指令的作用。

3.1.1 desc 指令

desc 指令是指令 description 的缩写。在查询表的数据时，如果对表的属性不了解，此时只要知道表的名字，就可以使用该指令获得该表的列属性信息。由于 SCOTT 用户是在创建数据库时系统默认设置的一个用户，该用户具有一定的权限和一些表，所以可以通过这些表联系 SQL 语句。在 SCOTT 用户下有一个表 dept，下面通过 desc 指令查询它的结构。

【实例 3-1】查询 SCOTT 用户中表 dept 的属性信息。

```
C:\>sqlplus /nolog
SQL*Plus: Release 11.1.0.6.0 - Production on 星期四 5月 14 17:15:10 2009
Copyright (c) 1982, 2007, Oracle.  All rights reserved..
SQL> connect scott/tiger
已连接。
SQL> desc dept;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ----------
 DEPTNO                                    NOT NULL NUMBER(2)
```

```
  DNAME                                        VARCHAR2(14)
  LOC                                          VARCHAR2(13)
```

这里的表结构是指表中的数据有哪些属性，这些属性直观地说就是表中的列的名字，如实例 3-1 中，表 dept 由三列组成，第一列为 DEPTNO（部门号），第二列为 DNAME（部门名），第三列为 LOC（部门所在地）。其中：

- 第一列 DEPTNO 的数据类型为整数，最大长度为 2。
- 第二列 DNAME 的数据类型为变长字符型，最大长度为 14 个字符。
- 第三列 LOC 的数据类型为变长字符型，最大长度为 13 个字符。



在实例 3-1 中给出了在 DOS 下启动 sqlplus 的方法，以及使用 sqlplus 工具、SCOTT 用户连接数据库的方法。connect 指令用于连接数据库，基本语法为：connect usrname/password@sid，因为是连接本机，故默认数据库不使用服务器名。

3.1.2 column 指令

顾名思义 column 是与列相关的指令。通过对列的输出值和列本身进行适当的格式化指令，使得表数据显示得更加人性化，也可以说，有了 column 指令使得 DBA 更容易通过 SQL*Plus 得到显示更加合理的表。首先给出该指令的语法格式：

```
col[umn] [{column|expr}[option...]]
```

这里 option 是指如下参数：

- FOR[MAT] format。
- CLE[AR]。
- HEA[DING]text。
- JUS[TIFY] {L[EFT]|C[ENTER] |R[IGHT]}：调整列在其显示长度内的位置。
- NEWL[INE]：从该列开始另起一列。
- NOPRI[NT]|PRI[NT]。
- NUL[L]text。
- ON|OFF：使得无法对该列进行其他格式化操作。

下面依次介绍这些指令的用法，并通过实例使得读者可以直观地理解如何使用 column 格式化指令。

1. FOR[MAT] format

首先查询表 salgrade 的全部信息。

（1）格式化模式 ‘9’

【实例 3-2】查看表 salgrade 中的全部数据。

```
SQL> SELECT *
```

```
  2  FROM salgrade;

     GRADE   LOSAL      HISAL
--------------- -------------- -----------------
         1        700       1200
         2       1201       1400
         3       1401       2000
         4       2001       3000
         5       3001       9999
```

显然该表中的三列：GRADE、LOSAL 和 HISAL 都占用了过多的宽度，使用如下的 column 指令格式化这些列的输出，使其占用较少的字符宽度。

【实例 3-3】格式化 number 类型数据（不带小数点）。

```
SQL> col grade for 99
SQL> col losal for 9999
SQL> col hisal for 9999
```

再次执行实例 3-2：

```
SQL> SELECT *
  2  FROM salgrad

GRADE LOSAL HISAL
---------- ---------- --------
     1   700  1200
     2  1201  1400
     3  1401  2000
     4  2001  3000
     5  3001  9999
```

显然该表中的三列：GRADE、LOSAL 和 HISAL 占用的宽度明显缩小，使用 column 指令格式化这些列的输出，使其占用较少的数位宽度。其中 GRADE 列占用 2 位数字的宽度，而 LOSAL 和 HISAL 各自占用 4 位数字的宽度。由于这三列的数据类型都为数字类型（NUMBER），所以使用 9 这样的选项，99 是格式化模式，每个 9 表示一位数字。

但是这里没有小数点，也不显示 0，那如何做到既显示小数点，又显示 0 呢？读者可以通过实例 3-4 体会。

【实例 3-4】格式化 NUMBER 类型数据（带小数点）。

```
SQL> col hisal for 9999.99
SQL> SELECT *
  2  FROM salgrade;

GRADE LOSAL    HISAL
---------- ---------- -------------
     1   700  1200.00
     2  1201  1400.00
     3  1401  2000.00
```

```
        4  2001  3000.00
        5  3001  9999.00
```

（2）格式化模式 ‘a’

如果显示结果的某列是字符型的，且占用较大宽度，此时可以使用格式化模式 ‘a’ 。

【实例 3-5】查看表 user_objects 中的部分信息。

```
SQL>SELECT object_name ,object_type
  2  FROM user_objects
  3  WHERE object type= 'TABLE'

OBJECT_NAME
---------------------------------------------------------------------------
OBJECT TYPE
------------------
BONUS
TABLE

DEPT
TABLE

DEPT_TEMP
TABLE
...............(省略了部分显示数据)
```

分析：从输出结果可以看出列 OBJECT_NAME 占用较多字符宽度，使得显示的数据很不协调，可以采用如下实例 3-6 所示的 column 指令格式化为 20 个字符宽度。

【实例 3-6】格式化字符类型数据。

```
SQL> col object_name for a20
```

重新执行查询指令如实例 3-7 所示。

【实例 3-7】格式化后再次查询 user_objects 中的部分数据。

```
SQL>SELECT object_name ,object_type
  2  FROM user objects
  3  WHERE object_type= 'TABLE'

OBJECT_NAME           OBJECT_TYPE
------------------------------------- ----------------------------------
BONUS                 TABLE
DEPT                   TABLE
DEPT TEMP             TABLE
EMP                    TABLE
EMP_TEMP              TABLE
ORD                    TABLE
PRODUCT               TABLE
SALGRADE              TABLE
SUPPLIER               TABLE
```

从输出可以看出，列 OBJECT_NAME 的宽度缩小了，宽度为 20 个字符，正因为该列是字符型数据，所以采用 20 来格式化，表示把字符型列的输出格式化为 20 个字符宽度。

（3）格式化模式 ‘$’

实例 3-4 的工资数据不是很合理，不知道是美元、英镑还是人民币，采用“$”格式化可以解决这个问题。

【实例 3-8】格式化货币。

```
SQL> col losal for $9999
SQL> col hisal for $9999
```

通过格式化，在使用 SQL 语句查询时，结果显示表的第二列数据头部都添加了“$”符号。

【实例 3-9】格式化货币后查询表 salgrade 中的所有数据。

```
SQL> SELECT *
  2  FROM salgrade;

    GRADE  LOSAL      HISAL
---------- ------ ----------
         1   $700      $1200
         2  $1201      $1400
         3  $1401      $2000
         4  $2001      $3000
         5  $3001      $9999
```

（4）格式化模式 ‘L’

如果数据库安装在不同的国家，如果希望显示本地的货币格式，该如何处理呢？SQL*Plus 使用了 L 格式化模式解决这个问题，如把实例 3-9 中表的第二列和第三列都改为本地货币格式。输入 column 格式化指令如实例 3-10 所示。

【实例 3-10】本地货币格式化。

```
SQL> col losal for L9999
SQL> col hisal for L9999
```

【实例 3-11】本地货币格式化后查询包 salgrade 的所有数据。

```
SQL> SELECT *
  2  FROM salgrade;
  GRADE             LOSAL             HISAL
-------------- ---------------------------- ----------------------
         1           RMB700        RMB1200
         2          RMB1201        RMB1400
         3          RMB1401        RMB2000
         4          RMB2001        RMB3000
         5          RMB3001        RMB9999
```

从输出结果可以看出，此时的货币单位是人民币。这是因为 column 命令的格式化模式“L”是显示本地货币的。其实，它是根据 Oracle 数据库的字符集来确定的，因为这里安装的数据库的

字符集为中文，所以显示的本地货币为人民币（RMB）。



读者可以使用如下的指令查询当前数据库支持的字符集。

```
SQL> SELECT userenv('language')
  2  FROM dual;

USERENV('LANGUAGE')
----------------------------------------------------
SIMPLIFIED CHINESE_CHINA.ZHS16GBK
```

其中 CHINESE_CHINA.ZHS16GBK 的组成是：语言_区域_字符集。显然这里的语言是中文，区域是中国，字符集是中文字符集。ZHS16GBK 的组成是：<语言><比特位><编码>。这里的语言是 ZHS，比特位是 16，编码方案是 GBK。字符集 ZHS16GBK 的含义是采用 GBK 编码的 16 位表示的简体中文。

2. CLE(AR)

如果不需要某列的格式化设置，可以采用实例 3-12 所示的 col 指令，用于删除在实例 3-11 中列 LOSAL 和 HISAL 的格式化设置。

【实例 3-12】删除列 LOSAL 和 HISAL 的格式化设置。

```
SQL> col LOSAL clear
SQL> col HISAL clear
```

为了验证是否删除列 LOSAL 和 HISAL 的格式化设置，可使用实例 3-13 查看这两列的格式化设置。

【实例 3-13】查看这两列的格式化设置。

```
SQL> col LOSAL
SP2-0046: COLUMN 'LOSAL' 未定义
SQL> col LOSAL
SP2-0046: COLUMN 'LOSAL' 未定义
```

显然，在实例 3-13 中删除列 LOSAL 和 HISAL 的格式化设置已成功执行。

3. HEA[DING] text

在实例 3-11 中，查询表 salgrade 时，列的属性如 LOSAL、HISAL 等显示得不是很直观，毕竟如果不同国家的人还是希望使用本国文字显示更加直观，再者，这些列的属性名字是程序员为了开发方便而起的“有意义”的名字，这些名字并非对所有人来说都容易理解，所以在 SQL*Plus 中给出了一个使用 HEA[DING]（方括号表示省略，不必写完整的单词）的格式化模式，如实例 3-14 所示。

【实例 3-14】使用 HEADING 格式化列 LOSAL 和 HISAL。

```
SQL> col LOSAL HEADING '低工资'
SQL> col HISAL HEADING '高工资'
```

利用实例 3-15 验证上述的修改是否成功。

【实例 3-15】验证实例 3-14 的格式化是否成功。

```
SQL> SELECT *
  2  FROM salgrade;

     GRADE          低工资           高工资
---------- --------------- ---------------
         1          RMB700         RMB1200
         2         RMB1201         RMB1400
         3         RMB1401         RMB2000
         4         RMB2001         RMB3000
         5         RMB3001         RMB9999
```

从实例 3-14 可以看出，列 LOSAL 和 HISAL 都已格式化，输出显示为容易理解的“低工资”和“高工资”，想必这样的表更容易接受吧！

4. JUS[TIFY] {L[EFT]|C[ENTER] |R[IGHT]}

JUS[TIFY] 的作用是调整列属性名字显示时在当前显示字符宽度范围内的位置，可以在字符宽度范围内的左边、中间和右边放置。先在实例 3-16 中查询表 dept。

【实例 3-16】查询表 dept 的全部数据。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
------------------ ------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

从结果可以看出，列 DEPTNO 显示在输出字符宽度的右边，而列 DNAME 和 LOC 显示在输出字符宽度的左边，下面调整列 DNAME 和 LOC，使得它们显示在其字符宽度的中间，如实例 3-17 所示。

【实例 3-17】格式化列 DNAME 和 LOC，使得列名显示在字符宽度的中间。

```
SQL> col dname jus center
SQL> col loc   jus center
```

利用实例 3-18 重新验证对列 DNAME 和 LOC 的格式化效果。

【实例 3-18】验证对列 DNAME 和 LOC 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO      DNAME             LOC
```

```
----------------- -------------------------- -----------------------
        10 ACCOUNTING    NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

通过实例 3-18 的格式化设置，使得列 DNAME 和 LOC 的显示位置在其字符宽度内得到调整。

4. NEWL[INE]

该格式化的作用是从该列开始的所有列的显示将另起一行，不过该格式化的用处不多，估计没有人愿意把表数据显示地如此凌乱吧。我们只通过实现让读者体会一下它的效果即可，如在格式化表 dept 中，使得在列 DNAME 以及其后的列属性名另起一行显示。

【实例 3-19】使用 NEWL[INE]格式化列。

```
SQL> col dname newline
```

利用实例 3-20 验证上述格式化的效果。

【实例 3-20】验证实例 3-19 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO
----------
DNAME          LOC
-------------- -------------
       10
ACCOUNTING     NEW YORK
```

显然，NEWLINE 的格式化得到验证，从列 DNAME 开始的所有列（DNAME、LOC）另起一行显示。在这种情况下，查看列中的数据不是很方便，只能沿着列名的位置向下寻找，显然让人很痛苦。在 SQL*Plus 中如果显示窗口的宽度不够，会自动另起一行显示，所以这个格式化指令用的很少。

5. NOPRI[NT]|PRI[NT

读者或许从英文字面就可以理解该指令的作用，NOPRI[NT]|PRI[NT]格式化使得格式化的列数据不显示（NOPRI）或者显示（PRI），对表 dept 的 LOC 列实现 NOPRI 格式化。

【实例 3-21】使用 NOPRI[NT]|PRI[NT 格式化。

```
SQL> col loc noprint
```

【实例 3-22】验证执行格式化结果。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME
------------------- --------------
       10  ACCOUNTING
```

```
        20  RESEARCH
        30  SALES
        40  OPERATIONS
        50  MARKETING
        60  MARK

已选择 6 行。
```

此时，列 LOC 的数据没有显示，如果打算恢复该列的显示可以使用实例 3-23 进行格式化。

【实例 3-23】恢复显示列 loc。

```
SQL> col loc print
```

这里的格式化效果留作读者验证。当然如果恢复该列的显示也可以使用 CLE[AR]清除格式化，这样就清除了所有对该列的格式化设置，如实例 3-24 所示。

【实例 3-24】清除对列 LOC 的所有格式化设置。

```
SQL> col loc clear
```

6. NUL[L] text

该格式化的目的是把列中值为空值的字段用随后的 text 文本代替，为了测试 NUL[L] text 格式化，首先使用实例 3-25 向表 dept 中插入一行数据。

【实例 3-25】向表 dept 中插入一行数据。

```
SQL> insert into dept (deptno,dname,loc)
  2           values(50,'Marcketing','');

已创建 1 行。
```

利用实例 3-26 验证是否插入数据。

【实例 3-26】验证实例 3-25 是否插入数据。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME          LOC
----------------- --------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing
```

从输出结果可以看出，已成功插入一行数据，DEPTNO 为 50，DNAME 为'MARKETING'，LOC 为空值（没有显示），使用实例 3-27 格式化，希望在输出时 LOC 为空值的字段显示为'TEMP'（表示临时）。

【实例 3-27】格式化列 LOC 使得该列为空值的字段显示为'TEMP'。

```
SQL> col loc null 'TEMP'
```

通过实例 3-28 验证格式化结果。

【实例 3-28】验证实例 3-27 格式化结果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
----------------- --------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing        TEMP
```

从显示结果可以看出，第 5 行记录中列 LOC 为空值的字段填充为'TEMP'。但这里只是显示给用户的数据发生了改变，而表中实际的数据没有任何变化。

7. ON/OFF

使用 OFF 格式化指令后，列的所有以前格式化将自动取消，以后任何的格式化该列虽然 SQL*Plus 不会报错，但是这些格式化修改都无效，如在实例 3-29 后继续操作，对表 dept 的列 LOC 实现 OFF 格式化。

【实例 3-29】使用 OFF 格式化列 LOC。

```
SQL> col loc off
```

再次验证格式化效果。

【实例 3-30】验证实例 3-29 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
------------------ -------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing
```

从结果可以看出，以前对列 LOC 的 NULL 格式化不再生效，下面重新对列 LOC 进行 NULL 格式化，并验证该格式化是否成功。

【实例 3-31】重新对列 LOC 进行 NULL 格式化。

```
SQL> col loc null 'temp'
SQL> SELECT *
  2  FROM dept;
```

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 Marcketing
```

可以看出在格式化时，系统没有报错，但是执行查询语句时，列 LOC 的 NULL 格式化没有生效。如果想继续对列 LOC 实现格式化操作，可使用实例 3-32 的 ON 格式化。

【实例 3-32】设置继续对列 LOC 实现格式化操作。

```
SQL> col loc on
```

3.1.3　run 或“/”指令

在使用 SQL*Plus 操纵 SQL 语句时，往往会重复执行 SQL 缓冲区中的语句，我们先执行一个查询，如实例 3-33 所示。

【实例 3-33】查询表 EMP 中的员工数据。

```
SQL> SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER';

EMPNO  ENAME      JOB        MGR HIREDATE          SAL
---------- ------------------------------- --------------------- -------
   7566  JONES      MANAGER    7839 02-4 月 -81         2975
   7698  BLAKE      MANAGER    7839 01-5 月 -81         2850
   7782  CLARK      MANAGER    7839 09-6 月 -81         2450
```

如果关于'MANAGER'的记录被改动或者其他原因需要继续执行该指令，则需要使用 run 或者“/”指令。通过实例 3-34 验证该指令，并注意二者的区别。

【实例 3-34】在实例 3-33 的查询后直接使用 run 和“/”指令。

```
SQL> run
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3* WHERE job = 'MANAGER'

EMPNO  ENAME      JOB        MGR HIREDATE          SAL
---------- ------------------- ------------------- ------ -------------
   7566  JONES      MANAGER    7839 02-4 月 -81         2975
   7698  BLAKE      MANAGER    7839 01-5 月 -81         2850
   7782  CLARK      MANAGER    7839 09-6 月 -81         2450

SQL> /

EMPNO  ENAME      JOB        MGR HIREDATE          SAL
```

```
---------- -------------------- -------------------- ------ -----------
   7566  JONES     MANAGER   7839  02-4月 -81         2975
   7698  BLAKE     MANAGER   7839  01-5月 -81         2850
   7782  CLARK     MANAGER   7839  09-6月 -81         2450
```

显然，在输入 run 时，SQL 语句由反馈显示出来，而“/”则没有 SQL 语句的反馈。其实使用 run 也可以不显示 SQL 语句的反馈信息，需要设置 SQL*Plus 的环境变量 FEEDBACK 来实现。

3.1.4　L(list)和 n 指令

L(list)指令用于列出当前 SQL 缓冲区中的 SQL 指令，在执行完实例 3-34 后，继续执行 L(list)指令，如实例 3-35 所示。

【实例 3-35】L(list)指令列出当前 SQL 缓冲区中的 SQL 指令语句。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3* WHERE job = 'MANAGER'未选定行

SQL>
```

注意此时在第 3 行有一个“*”号，因为 SQL*Plus 允许修改 SQL 缓冲区中的 SQL 语句，如更改第 1 行，不需要查询 sal 列等信息。“*”号就定位了要修改的行，实例 3-36 演示了如何定位要修改的行。

【实例 3-36】定位要修改的行。

```
SQL> 1
  1* SELECT empno ,ename,job,mgr,hiredate,sal
SQL>
```

一旦输入行号 1，则提示定位在第一行。

3.1.5　change 和 n (next)指令

执行完实例 3-36 后继续执行 ch(ange)指令，用于修改某行的字段，如实例 3-37 所示。

【实例 3-37】修改第 1 行的字段。

```
SQL> ch /sal/deptno
  1* SELECT empno ,ename,job,mgr,hiredate,deptno
SQL> /

EMPNO ENAME      JOB          MGR HIREDATE       DEPTNO
----- ---------- --------- ----- ---------- ----------
 7566 JONES      MANAGER    7839 02-4月 -81          20
 7698 BLAKE      MANAGER    7839 01-5月 -81          30
 7782 CLARK      MANAGER    7839 09-6月 -81          10
```

使用 change 指令把列 sal 修改为列 deptno，一旦修改成功则随后显示修改结果。n 指令用来修改整行 SQL 语句，n 为在 SQL 语句中的行号，随后输入的语句为替换 n 行的 SQL 语句。在实例 3-37 执行成功后，执行实例 3-38 来查看修改结果。

【实例 3-38】通过 list 指令查看实例 3-37 的修改结果。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,deptno
  2  FROM emp
  3* WHERE job = 'MANAGER'
SQL> 1 SELECT empno,ename,job,mgr
SQL> list
  1  SELECT empno,ename,job,mgr
  2  FROM emp
  3* WHERE job = 'MANAGER'
SQL>
```

首先输入 list 指令，查询当前 SQL 缓冲区中的语句，输入“1 SELECT empno,ename,job,mgr”表示把第一行的 SQL 语句替换为“SELECT empno,ename,job,mgr”，执行后，输入 list 再次验证，发现修改成功。

3.1.6　附加(a)指令

附加(a)指令就是在某行末尾添加一些语句或属性信息，如通过实例 3-39 来查询 demp 表中所有部门的名字。

【实例 3-39】查询 demp 表中所有部门的名字。

```
SQL> SELECT dname
  2  FROM dept;

DNAME
--------------
ACCOUNTING
RESEARCH
SALES
OPERATIONS
Marcketing
```

如果想同时知道每个部门的所在地，可使用如下方式修改 SQL 缓冲区中的语句，如实例 3-40 所示。

【实例 3-40】用 list 指令查询当前的指令。

```
SQL> list
  1  SELECT dname
  2* FROM dept
```

显然，此时“*”号在第二行，而不是我们要修改的第一行。

【实例 3-41】定位要修改的行。

```
SQL> 1
  1* SELECT dname
```

此时，“*”号定位在第一行，说明已经将 SQL 缓冲区中的第一行语句设为当前行。接着可以使用 a（附加）指令将“，loc”添加在第一行 SQL 语句的末尾了。

【实例 3-42】使用 a（附加）指令将“，loc”添加在第一行 SQL 语句的末尾。

```
SQL> a ,loc
  1* SELECT dname,loc
```

再次使用 list 指令验证修改效果：

```
SQL> list
  1  SELECT dname,loc
  2* FROM dept
```

显然，修改成功，在 SQL 语句的第一行添加了列 LOC，此时可使用 run 命令继续执行，以输出希望的数据。

【实例 3-43】使用 run 指令查看实例 3-42 的修改结果。

```
SQL> run
  1  SELECT dname,loc
  2* FROM dept

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
Marcketing
```

3.1.7 del 指令

del 指令用于删除 SQL 缓冲区中的某行 SQL 语句。其语法格式是：del n（n 为行号），所以在删除某行之前需要做一些工作，首先需要输入 list 指令验证当前 SQL 缓冲区中的 SQL 语句，确定要删除的行，执行 del n 指令，而后输入 list 指令验证删除结果。

为了验证整个过程，可输入实例 3-44 的语句。

【实例 3-44】执行 SQL 查询。

```
SQL> SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4  order by sal;
```

【实例 3-45】使用 list 指令验证当前 SQL 缓冲区中的 SQL 语句。

```
SQL> l
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4* order by sal
```

如要删除第 3 和第 4 行，过程如实例 3-46 所示。

【实例 3-46】删除当前 SQL 缓冲区中的 SQL 语句的行。

```
SQL> del 4
SQL> del 3
```

【实例 3-47】使用 list 指令查询实例 3-46 删除后的 SQL 指令。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2* FROM emp
```

从输出结果可以看出，已成功删除了 SQL 语句的第 3 和第 4 行。再通过实例 3-48 验证修改后的结果。

【实例 3-48】验证实例 3-46 的删除结果。

```
SQL> /

EMPNO ENAME      JOB         MGR HIREDATE         SAL
---------- -------------------- ---------------- --------- -----------------
  7369 SMITH      CLERK       7902 17-12 月-80        800
  7499 ALLEN     SALESMAN    7698  20-2 月 -81       1600
  7521 WARD      SALESMAN   7698  22-2 月 -81       1250
  7566 JONES      MANAGER    7839  02-4 月 -81        2975
  7654 MARTIN    SALESMAN   7698  28-9 月 -81       1250
  7698 BLAKE     MANAGER    7839  01-5 月 -81       2850
  7782 CLARK     MANAGER    7839  09-6 月 -81       2450
  7839 KING       PRESIDENT        17-11 月-81        5000
  7844 TURNER    SALESMAN   7698  08-9 月 -81       1500
  7900 JAMES     CLERK       7698  03-12 月-81        950
  7902 FORD      ANALYST    7566  03-12 月-81        3000

EMPNO ENAME      JOB         MGR HIREDATE         SAL
---------- -------------------- ---------------- --------- -------------------
  7934 MILLER    CLERK        7782  23-1 月 -82           1300

已选择 12 行。
```

3.1.8　set line 指令

该指令的使用格式为：set line {80/n}，作用是将查询的数据输出设置为 n 个字符宽显示。默认

是以 80 个字符的宽度输出。如果一个输出由于显示宽度不够使得有的列错行，这样的数据输出就很难让人接受。实例 3-49 用于查询 SCOTT 用户下的 EMP 表的全部信息。

【实例 3-49】查询表 EMP 的全部数据。

```
SQL> SELECT *
  2  FROM emp;

    EMPNO ENAME      JOB        MGR HIREDATE      SAL       COMM
------------------- ------------------- -------------- ------- -------
   DEPTNO
------------------
     7369 SMITH      CLERK        7902  17-12 月-80      800
       20

     7499 ALLEN   SALESMAN       7698  20-2 月 -81       1600        300
       30
..............（省略了部分数据）
```

此时，采用默认的 80 个字符宽度，显然这样的宽度是不够的。使用 set line n 指令设置输出的显示字符宽度为 100 个字符，改善数据的显示，如实例 3-50 所示。

【实例 3-50】使用 SET LINE 格式化行的长度并继续查询。

```
SQL> set line 100
SQL> /
EMPNO ENAME      JOB      MGR HIREDATE     SAL     COMM     DEPTNO
---------- ---------- ---------------- ---------- ---------------- ----------
  7369 SMITH    CLERK      7902  17-12 月-80     800                20
  7499 ALLEN   SALESMAN  7698  20-2 月 -81     1600       300       30
  7521 WARD    SALESMAN  7698  22-2 月 -81     1250       500       30
  7566 JONES    MANAGER   7839  02-4 月 -81     2975                20
  7654 MARTIN  SALESMAN   7698  28-9 月 -81     1250      1400       30
..............（省略了部分数据）
```

使用 set line 100 指令将以后显示的行的长度设置为 100 个字符，再次查询表 EMP 中的全部数据，此时不会出现如实例 3-49 所示的拐行的现象了。

3.1.9 spool 指令

spool 指令的作用是把用户输入的 SQL 语句和查询结果存储在指定的文件中，下面通过实例来介绍如何使用 spool 指令。

【实例 3-51】查看 spool 的状态。

```
SQL> show spool
spool OFF
```

这里使用 SHOW 查看 SQL*Plus 的参数 spool 的状态，该指令为关闭状态，那么如何开启并使用 spool 功能呢？实例 3-52 说明了用法。

【实例 3-52】启动 spool 并将查询记录到.txt 文件中。

```
SQL> spool d:\spool_test
SQL> show spool
spool ON
```

这个实例的作用是开启 spool 并把接下来用户输入的 SQL 语句和查询结果存储到指定的 d:\spool_test 文件中，查看参数 spool 的状态为 ON。

【实例 3-53】执行查询语句。

```
SQL>  SELECT empno, ename, job, mgr, sal
  2   FROM emp
  3   WHERE job = 'MANAGER';

EMPNO  ENAME      JOB         MGR     SAL
---------- ----------------- -------------------- -------------- -----------
  7566  JONES      MANAGER    7839      2975
  7698  BLAKE      MANAGER    7839      2850
 7782  CLARK      MANAGER    7839      2450
```

【实例 3-54】关闭 spool 功能。

```
SQL> spool off
```

执行完实例 3-53 和 3-54 后，在 D 盘的根目录下生成一个 spool_test.lst 文件，用记事本打开该文件，内容如图 3-1 所示。

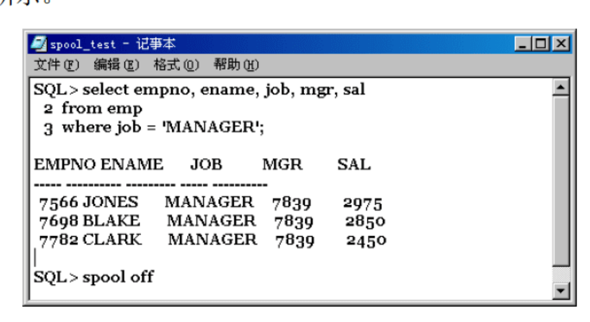


图 3-1　spool_test 文件内容



当用户查询并输出大量的数据时，为了方便可以使用该指令，这样既可以保存输出记录，也方便使用记事本的工具查找相应的数据。

3.2　控制 SQL*Plus 工具的环境

3.2.1　ECHO 环境变量

SQL*Plus 的 ECHO 环境变量有两种状态，即 ON 和 OFF。如果 ECHO 环境变量的状态为打开，

则使用脚本文件执行 SQL 语句时，脚本文件中的 SQL 语句会输出显示，否则，不显示脚本文件中的 SQL 语句。

查看该变量的状态如实例 3-55 所示。

【实例 3-55】查询 ECHO 状态。

```
SQL> show echo
echo OFF
```

因为当前的 ECHO 环境变量的状态是 OFF，所以不会显示脚本文件中的 SQL 语句。

【实例 3-56】在 echo 关闭状态下执行查询脚本。

```
SQL> @d:\SELECT emp

     EMPNO ENAME      JOB           MGR HIREDATE       SAL
------------------ ----------- ------------------- ---------------- ----
      7782 CLARK      MANAGER      7839 09-6 月 -81       2450
      7698 BLAKE      MANAGER      7839 01-5 月 -81       2850
      7566 JONES      MANAGER       7839 02-4 月 -81       2975
```

此时，将 ECHO 环境变量的状态设置为 ON。

【实例 3-57】将 ECHO 环境变量的状态设置为 ON。

```
SQL> set echo on
```

通过实例 3-58 查看 ECHO 环境变量的当前状态。

【实例 3-58】查看 ECHO 环境变量的当前状态。

```
SQL> show echo
echo ON
```

现在使用实例 3-59 测试执行脚本文件时，是否显示脚本文件中的 SQL 指令。

【实例 3-59】在 ECHO 打开状态下执行查询脚本。

```
SQL> @d:\SELECT_emp
SQL> SELECT empno, ename, job, mgr, hiredate, sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4  order by sal
  5  /

     EMPNO ENAME      JOB           MGR HIREDATE       SAL
------------------ ----------- ------------------- ---------------- ---
      7782 CLARK      MANAGER      7839 09-6 月 -81       2450
      7698 BLAKE      MANAGER      7839 01-5 月 -81       2850
      7566 JONES      MANAGER       7839 02-4 月 -81       2975
```

测试结果表明，当 ECHO 环境变量的当前状态为 ON 时执行脚本文件，显示脚本文件中的 SQL 指令。

3.2.2　FEEDBACK 环境变量

FEEDBACK 环境变量用于控制查询输出的数据行数是否显示，以及记录数达到什么值时才显示数据行数。其语法格式为：SET FEED[BACK]{6/n/on/off}。

使用实例 3-60 查看 FEEDBACK 变量的当前值。

【实例 3-60】查看 FEEDBACK 变量的当前值。

```
SQL> show feedback
用于 6 或更多行的 FEEDBACK ON
```

输出结果表示当数据行数等于或多于 6 行时，才显示数据行数，实例 3-61 用于查询表 salgrade 的数据。

【实例 3-61】查询表 salgrade 的数据。

```
SQL> SELECT *
  2  FROM salgrade;

    GRADE      LOSAL      HISAL
---------------- --------------------- -----------------
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999

已选择 10 行。
```

显然，因为输出数据行数大于 6，所以显示计算得到的数据行数。也可以更改变量 FEEDBACK 的参数，如实例 3-62 所示。

【实例 3-62】更改变量 FEEDBACK 的参数。

```
SQL> set feedback 12
```

为了验证修改是否成功，可使用实例 3-63 查询当前 FEEDBACK 变量的状态。

【实例 3-63】查询当前 FEEDBACK 变量的状态。

```
SQL> show feedback
用于 12 或更多行的 FEEDBACK ON
```

显然修改成功，此时为了测试修改结果，再次执行实例 3-61，结果如下：

```
    GRADE      LOSAL      HISAL
----------------- ------------------- ------------------
```

```
          1        700       1200
          2       1201       1400
          3       1401       2000
          4       2001       3000
          5       3001       9999
          1        700       1200
          2       1201       1400
          3       1401       2000
          4       2001       3000
          5       3001       9999
```

因为当前的 FEEDBACK 参数为 12，所以小于 12 行的数据行数是不显示的。

3.3 本章小结

本章主要讲解了 SQL*Plus 工具，该工具是 Oracle 提供的用户操作数据库的人机接口，通过这个接口，用户可在拥有一定权限的条件下操纵数据库、更改数据库的各种系统配置。SQL*Plus 工具作为一个应用工具软件，提供了一系列的编辑指令，使得用户可更方便地操作和使用 SQL*Plus 工具。如果用户经常使用同样的 SQL 语句，为了提高输入效率可以使用 SQL*Plus 提供的脚本文件。为方便显示用户数据和其他信息，SQL*Plus 工具还提供了环境变量。

通过本章的学习，读者应该能够熟练使用 SQL*Plus 的各种编辑指令和格式化指令，掌握如何编辑和运行脚本文件，这些都是日常维护中必备的技能。

第 4 章

◂ SQL语言概述 ▸

SQL 语言是“结构化查询语言”的缩写，即 Structured Query Language。两个工业界认可的国际机构 ANSI 和 ISO 把 SQL 作为关系数据库的标准语言。SQL 语言涉及的语句简单，语义明了，使用该语言可检索和维护数据库、编写涉及数据库操作的应用程序或脚本语言。在实际工作中，我们经常使用数据查询语句和数据操纵语句。在使用这些 SQL 语句时，可以使用一些函数来处理输出结果或者通过分组函数使得输出数据更加友好。

4.1 SQL 的语句分类

SQL 语句按照其功能可分为 5 类，即数据查询语句、数据操纵语句、数据定义语句、事务控制语句和数据控制语句。下面通过 SQL 语句关键字依次简单介绍这些语句的功能，在本书后续的章节中读者将学到如何使用这些语句，以及使用这些语句的场合。

1. 数据查询语句

SELECT：该语句的功能是从数据库中获得用户数据，如查询一个表中的全部数据等。

2. 数据操纵语句

对该语句的说明如下。

- INSERT：该语句的功能是向表中添加记录。
- UPDATE：该语句的功能是更新表中的数据，通常和 WHERE 条件语句一起使用。
- DELETE：删除表中的数据。

3. 数据定义语句

对该语句的说明如下。

- CREATE：创建数据库对象，如表、索引、视图等。
- ALTER：改变系统参数，如改变 SGA 的大小等。
- DROP：删除一个对象，如删除一个表、索引或者序列号等。
- RENAME：重命名一个对象。

- TRUNCATE：截断一个表。

4. 事务控制语句

对该语句的说明如下。

- COMMIT：用于提交由 DML 语句操作的事务。
- ROLLBACK：用于回滚 DML 语句改变了的数据。

5. 数据控制语句

对该语句的说明如下。

- GRANT：用于授予用户访问某对象的特权。
- REVOKE：用于回收用户访问某对象的特权。

本章将重点介绍数据查询语句和数据操纵语句，数据定义语句、事务控制语句、数据控制语句可参考后面的章节内容。

4.2 数据查询语句

Oracle 的 SQL 查询语句，即 SELECT 语句。如果需要检索数据库中的数据，就需要使用该语句。在使用 SELECT 语句时，必须有相应的 FROM 子句。当需要复杂查询时可以使用 WHERE 子句。把整个查询语句中的 SELECT、FROM 和 WHERE 称为关键字，下面详细介绍查询语句的用法，并给出常用的运算符用法。

4.2.1 关键字概述

1. SELECT

一个简单的 SELECT 语句至少包含一个 SELECT 子句和一个 FROM 子句。其中 SELECT 子句指明要显示的列，而 FROM 子句指明包含要查询的表，该表包含了在 SELECT 子句中的列。其语法格式如下。

```
SELECT *|{[DISTINCT] column | expression [alias],…}
FROM    table;
```

在上述语法规则中，“|”号表示或的关系，“[]”表示可选。

其中：

- SELECT：选择一个列或多个列。
- *：选择表中所有的列。
- DISTINCT：去掉列中重复的值。
- column|expression：选择列的名字或表达式。
- alias：为指定的列设置不同的标题。

- FROM table：指定要选择的列所在的表，即对该表进行数据检索。

上面涉及到的语法，在下文都会介绍。其中有几个术语需要读者分辨清楚，因为在接下来的内容中将多次用到，它们是关键字、子句和语句。

- 关键字：它是一个单独的 SQL 元素，如 SELECT、FROM 等都是关键字，并且要求关键字不能简写，如写成 SEL、FRO 是不允许的，但是不要求必须大写，大写是 Oracle 推荐的写法，即关键字都大写而其他小写，以做区分。
- 子句：子句是 SQL 语句的一部分，它不是一个可执行的 SQL 语句，如“SELECT *”就是一个子句。
- 语句：语句由一个或多个子句组成，它是可执行的，如“SELECT * FROM dept”就是一个语句。在书写语句时，读者最好采取每个子句一行的习惯，这样可增强可读性。

（1）简单查询

先使用一个实例说明如何实现一个简单的查询，此时我们使用 SCOTT（该用户在创建数据库时会自动创建）的 dept 表，如实例 4-1 所示。

【实例 4-1】查询 SCOTT 用户的 dept 表的全部内容。

```
SQL> conn scott/tiger
已连接。
SQL> SELECT *
  2  FROM dept;

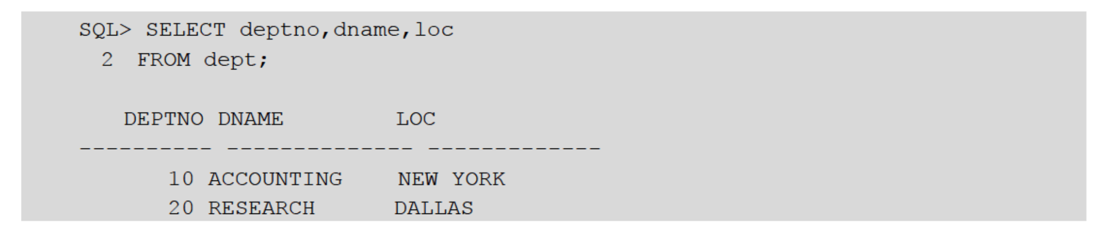
   DEPTNO DNAME          LOC
---------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
```

首先使用 SCOTT 用户登录，该用户的默认密码是 TIGER，此时不区分用户名和密码的大小写，然后输入一个查询语句，该语句的作用是查询表 dept 中的所有列的数据。

这里“*”号的含义是选择表中的所有列，FROM 关键字后是表名。dept 是一个部门表，该表有三列，分别是 DEPTNO（部门号）、DNAME（部门名称）和 LOC（部门所在地）。

还有一种查询方式用于实现查询表中的所有列的数据，即在 SELECT 关键字后输入所有列的名字，名字之间用逗号分开，如在实例 4-2 中重新查询表 dept 中的所有列的数据。

【实例 4-2】重新查询表 dept 的全部内容。



```
SQL> SELECT deptno,dname,loc
  2  FROM dept;

   DEPTNO DNAME          LOC
---------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
```

```
        30 SALES          CHICAGO
        40 OPERATIONS      BOSTON
```

在实例 4-2 中，SELECT 关键字之后的列名用逗号分开。



实例 4-1 和实例 4-2 都是操作用户 SCOTT 的表，若想操作顺利，需要读者使用 SCOTT 登录，如果使用 SYSTEM 登录，就会提示错误。

【实例 4-3】使用 SYSTEM 用户登录，该用户的默认密码是 MANAGER。

```
SQL> conn system/manager
已连接。
SQL> SELECT *
  2  FROM dept;
FROM dept
     *
ERROR 位于第 2 行:
ORA-00942: 表或视图不存在
```

此时如果将 FROM 子句改为 FROM scott.dept，该语句就会顺利执行，如实例 4-4 所示。

【实例 4-4】在 SYSTEM 用户模式下使用“模式名.表名”的方式查询表数据。

```
SQL> conn system/manager
已连接。
SQL> SELECT *
  2  FROM scott.dept;

   DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

因为 SYSTEM 用户为系统管理员，所以他有权限操作 SCOTT 用户的对象，使用 SYSTEM 用户登录，在查询用户 SCOTT 的对象时只需要在对象前指明是该用户的对象即可。

（2）特定的列查询

在实际操作中，并不是表中所有的列都需要查询，此时只需要在 SELECT 关键字后输入要查询的列名即可实现对特定列的查询，如实例 4-5 所示。

【实例 4-5】查询表 dept 的特定列的数据。

```
SQL> SELECT dname,loc
  2  FROM dept;

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
```

```
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

其实，在 SELECT 之后可以输入表中存在的任意列，并且列的顺序没有要求，数据的显示将以用户输入的列的顺序为基准，如实例 4-6 所示。

【实例 4-6】查询表 dept 中的任意列的数据。

```
SQL> SELECT dname,loc,deptno
  2  FROM dept;

DNAME          LOC            DEPTNO
-------------- ------------- ----------
ACCOUNTING     NEW YORK          10
RESEARCH       DALLAS            20
SALES          CHICAGO           30
OPERATIONS     BOSTON            40
```

（3）列的别名查询

在使用 SELECT 语句时，SQL*Plus 使用选择的列名作为列标题，并且采用大写方式。由于表中的列名是数据库开发人员或程序员设计的，是为了编程的需要，但是这样的列标题可能不具备描述性，从而难以理解，Oracle 提供了列别名来更改列标题的显示方式，如实例 4-7 所示。

【实例 4-7】通过别名更改列标题的查询。

```
SQL>SELECT empno,ename employee_name,sal AS salary,deptno "Deptmentnumber"
  2* FROM emp

 EMPNO EMPLOYEE_NAME           SALARY Deptmentnumber
-------------- ------------------------------------------- ---------- ----
     7369 SMITH                     800           20
     7499 ALLEN                    1600           30
     7521 WARD                     1250           30
     7566 JONES                     2975           20
     7654 MARTIN                   1250           30
     7698 BLAKE                     2850           30
     7782 CLARK                    2450           10
     7839 KING                      5000           10
     7844 TURNER                   1500           30
     7900 JAMES                      950           30
     7902 FORD                      3000           20

  EMPNO EMPLOYEE_NAME           SALARY Deptmentnumber
-------------- ------------------------------------------- ----------
     7934 MILLER                   1300           10

已选择 12 行。
```

创建别名时，在列名后使用 AS 关键字，之后紧跟别名，或者在列名后加空格，然后紧接着是别名，如上例中 ename employee_name 和 sal AS salary，但是此时的列标题在显示时为别名的大写格式。如果要保持别名的格式，可以使用双引号，此时的列标题就会如别名的格式，如 deptno "Deptmentnumber"。

2. WHERE

如果用户想查询一个特定条件的表该怎么办呢？Oracle 提供了 WHERE 子句来限制查询条件，WHERE 子句可以限制选择的行数。这样可以实现满足一定条件的数据查询，从而实现更加灵活的应用。实例 4-8 用于查询 SCOTT 用户的表 dept 中满足部门驻地在 CHICAGO 的部门所有信息。

【实例 4-8】使用 WHERE 子句查询 SCOTT 用户的表 dept 的全部数据。

```
SQL> SELECT *
  2  FROM dept
  3  WHERE loc = 'CHICAGO';

   DEPTNO DNAME          LOC
----------------- --------------------- -------------
       30  SALES          CHICAGO
```

上述查询虽然使用了“SELECT *”子句，但是 WHERE 子句限制了查询的结果必须是 LOC 为 CHICAGO 的部门信息，所以限制了查询的行数。当然，用户也可以输入其他限制性条件，如查询部门号小于 30 的部门信息，如实例 4-9 所示。

【实例 4-9】查询表 dept 中部门号小于 30 的所有数据。

```
SQL> SELECT *
  2  FROM dept
  3  WHERE deptno < 30;

   DEPTNO DNAME          LOC
---------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
```

在 WHERE 子句中的条件可以根据需要，通过各种算数或逻辑运算符实现条件限制。对 WHERE 子句更加详细地讲解将在第 10 章数据查询中介绍。

在上面介绍的查询语句显示的结果都是 Oracle 提供的，我们并没有对显示的信息做任何的修改，即数据的显示结果是 Oracle 的默认结果。在 Oracle 中列标题的显示满足如下规则：

- 字符和日期型的列标题显示在宽度的左边。
- 数字型的列标题显示在宽度的右边。
- 默认的列标题都是大写的，如图 4-1 所示。

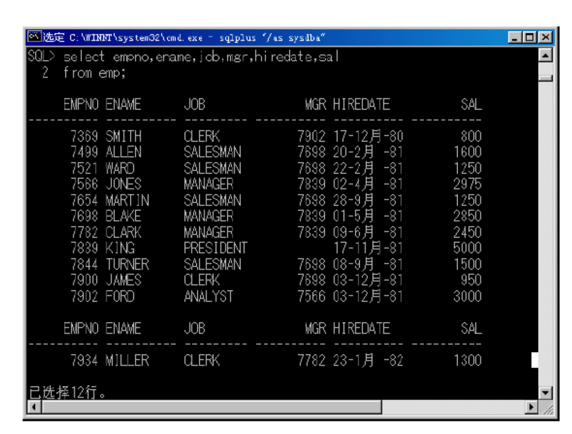


图 4-1　列标题的默认属性

4.2.2　使用运算符

通常情况下，SQL 语句中多配合使用运算符来达到对预期项的查询，如算数运算符、DISTINCT 运算符、连接运算符等。

1. 算数运算符

算数运算符，即加、减、乘、除 4 种运算：+、-、*、/。使用算数运算符实现对日期型和数字型的算数操作。创建一个具有算数运算的表达式，丰富查询的显示结果，如在 SCOTT 用户的 EMP 表中，查询每个员工的年薪，如实例 4-10 所示。

【实例 4-10】查询 SCOTT 用户 EMP 表中员工的名字和年薪。

```
SQL> SELECT ename "员工姓名",sal*12 "年薪"
  2  FROM emp
  3  WHERE job = 'MANAGER';

员工姓名             年薪
-------------------- ----------
JONES                35700
BLAKE                34200
CLARK                29400
```

其他运算符的使用规则类似，读者可以自行测试，如为所有 job= "SALESMAN"的员工月薪增加 500。

算数运算符遵循一定的优先顺序，即“乘除”优先于“加减”，而“乘除”具有同等优先权，“加减”也具有同等优先权。同等优先权的运算符按照从左到右的顺序计算，如 sal*12＋1000，先计算 sal*12，再加 1000 就是该表达式的最后计算结果，如实例 4-11 所示。

【实例 4-11】在查询中使用运算符。

```
SQL>SELECT ename "员工姓名",sal*12+1000 "年薪"
  2  FROM emp
```

```
  3* WHERE job = 'MANAGER'

员工姓名             年薪
---------- ----------
JONES            36700
BLAKE            35200
CLARK            30400
```

2. DISTINCT 运算符

DISTINCT 运算符用于使查询的结果没有重复内容，如需要查询 SCOTT 用户的 EMP 表中有多少个 job。先用实例 4-12 测试不使用 DISTINCT 的查询结果，再使用实例 4-13 测试使用 DISTINCT 的查询结果，通过两个结果对比，读者可以清晰体会使用 DISTINCT 的区别。

【实例 4-12】查询表 EMP 中的 JOB 名。

```
SQL> SELECT job
  2  FROM emp;

JOB
---------
CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
PRESIDENT
SALESMAN
CLERK
ANALYST

JOB
---------
CLERK

已选择 12 行。
```

在实例 4-12 中，选择结果有 12 行，重复的 JOB 内容也会显示在结果中，显然这样的结果不是我们想要的，而实例 4-13 使用 DISTINCT 关键字实现不重复查询。

【实例 4-13】使用 DISTINCT 关键字实现不重复查询表 emp 中的 job 名。

```
SQL> SELECT distinct job
  2  FROM emp;

JOB
---------
ANALYST
CLERK
```

```
MANAGER
PRESIDENT
SALESMAN
```

在 SELECT 关键字后紧跟 DISTINCT 关键字，使得选择的行没有重复的结果，但是如果 DISTINCT 关键字后有多个列，情况如何呢？如实例 4-14 所示。

【实例 4-14】使用 DISTINCT 关键字实现多列查询。

```
SQL> SELECT distinct deptno,job
  2  FROM emp;

  DEPTNO JOB
--------------- ----------------
       10 CLERK
       10 MANAGER
       10 PRESIDENT
       20 ANALYST
       20 CLERK
       20 MANAGER
       30 CLERK
       30 MANAGER
       30 SALESMAN

已选择 9 行。
```

此时使用 DISTINCT 关键字使得结果中多个列的组合没有重复的结果，即每一行数据不完全相同。

3. 连接运算符

连接运算符把列与其他列连接起来，也可以把列与字符串连接起来。连接符是两个竖线“||”，在连接字符串时使用单引号''。实例 4-15 是使用连接运算符的实例。

【实例 4-15】使用连接运算符“||”。

```
SQL> SELECT ename  ||' is a '||job ||' and 1month salary is:'|| sal  As "The implyees's information"
  2  FROM emp;

The implyees's information
---------------------------------------------------------------------------
SMITH is a CLERK and 1month salary is:800
ALLEN is a SALESMAN and 1month salary is:1600
WARD is a SALESMAN and 1month salary is:1250
JONES is a MANAGER and 1month salary is:2975
MARTIN is a SALESMAN and 1month salary is:1250
BLAKE is a MANAGER and 1month salary is:2850
CLARK is a MANAGER and 1month salary is:2450
KING is a PRESIDENT and 1month salary is:5000
TURNER is a SALESMAN and 1month salary is:1500
```

```
JAMES is a CLERK and 1month salary is:950
FORD is a ANALYST and 1month salary is:3000

The imployees's information
--------------------------------------------------------------------------
MILLER is a CLERK and 1month salary is:1300

已选择 12 行。
```

该实例中使用了 4 个连接运算符把三列（ename、job 和 sal）和两个字符串（"is a"和"and 1 month salary is:"）连接起来。显然这样的显示结果更容易阅读。在上例中，我们也使用了别名，即将显示的信息设置一个列标题为 "The imployees's information"。

通过以上介绍，可总结出 SQL 语句的书写规范，在书写 SQL 语句时，不区分大小写，如 SELECT 和 SELECT 都是允许的。但是关键字不能跨行书写，也不能缩写，如 SELECT 不能写成 SEL。一个 SQL 语句可以有多行，Oracle 推荐了书写 SQL 语句的规范，使用该规范会使得 SQL 语句更加容易阅读，并且容易区分 SQL 语句的关键字和其他对象名。推荐的规范如下。

- SQL 语句的关键字要大写，对象名小写。
- 缩进对齐，这样便于阅读。
- 每个子句一行。

【实例 4-16】查询表 EMP 中工资大于 1500 的员工信息。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE  sal > 1500;

     EMPNO ENAME      JOB              MGR HIREDATE          SAL
---------- ---------- --------- ---------- ---------- ----------
      7499 ALLEN      SALESMAN        7698 20-2 月 -81       1600
      7566 JONES      MANAGER         7839 02-4 月 -81       2975
      7698 BLAKE      MANAGER         7839 01-5 月 -81       2850
      7782 CLARK      MANAGER         7839 09-6 月 -81       2450
      7839 KING       PRESIDENT            17-11 月-81       5000
      7902 FORD       ANALYST         7566 03-12 月-81       3000

已选择 6 行。
```

每个子句一行使得阅读方便，通过 SQL 关键字大写，使得它们与 Oracle 对象区分开来，这样整个代码就很清晰了。

4.2.3 使用单行函数

为了方便数据库的操作，Oracle 提供了各种函数操作，单行函数分为字符型单行函数、数字型单行函数和日期型单行函数。

1. 字符型单行函数

字符型单行函数接收一个字符输入，并且返回一个计算结果，该结果可以是字符型，也可以是数字型。常用的单行字符型函数如下。

（1）LOWER

其函数的格式为：LOWER(column | expression)，函数功能是把字符串转换成小写，如实例 4-17 所示。

【实例 4-17】使用单行函数 LOWER()。

```
SQL> SELECT LOWER('Structured Query Language')
  2  FROM dual;

LOWER('STRUCTUREDQUERYLAN
-------------------------
structured query language
```

（2）UPPER

其函数格式为：UPPER(column | expression) ，函数功能是把字符串转换成大写，如实例 4-18 所示。

【实例 4-18】使用单行函数 UPPER()。

```
SQL> SELECT UPPER('Structured Query Language')
  2  FROM dual;

UPPER('STRUCTUREDQUERYLAN
-------------------------
STRUCTURED QUERY LANGUAGE
```

（3）INITCAP

其函数格式为：INITCAP(column | expression)，其功能是把字符串的首字母大写，如实例 4-19 所示。

【实例 4-19】使用单行函数 INITCAP()。

```
SQL> SELECT INITCAP('structured query language')
  2  FROM dual;

INITCAP('STRUCTUREDQUERYL
-------------------------
Structured Query Language
```

（4）CONCAT

其函数格式为：CONCAT(column1 | expression1, Column2 | expression2)，该函数用于连接两个字符串，或者连接两个列中的数据。

【实例 4-20】使用单行函数 CONCAT()。

```
SQL> SELECT CONCAT ('Structured Query Language','is easy to learn!')
```

```
  2  FROM dual;

CONCAT('STRUCTUREDQUERYLANGUAGE','ISEASYTO
------------------------------------------
Structured Query Languageis easy to learn!
```

在函数 CONCAT 中，参数也可以是列名，列名和表达式可以根据需要自由选择，如实例 4-21 所示。

【实例 4-21】使用列名和表达式的 CONCAT 函数。

```
SQL> SELECT concat(ename,hiredate) "emp_name,hiredate"
  2  FROM emp;

emp name,hiredate
--------------------
SMITH17-12 月-80
ALLEN20-2 月 -81
WARD22-2 月 -81
JONES02-4 月 -81
MARTIN28-9 月 -81
BLAKE01-5 月 -81
CLARK09-6 月 -81
SCOTT19-4 月 -87
KING17-11 月-81
TURNER08-9 月 -81
ADAMS23-5 月 -87

emp name,hiredate
--------------------
JAMES03-12 月-81
FORD03-12 月-81
MILLER23-1 月 -82

已选择 14 行。
```

（5）SUBSTR

其函数格式为：SUBSTR(column | expression,m [,n])，该函数从一个字符串中获取一个子串，该子串从 expression 的第 m 个字符开始，到第 n 个字符结束，如果不指定 n，则从第 m 个字符开始到 expression 表达式的结尾，如实例 4-22 所示。

【实例 4-22】使用 SUBSTR 函数。

```
SQL> SELECT SUBSTR('structured query language',12)
  2  FROM dual;

SUBSTR('STRUCT
--------------
query language
```



函数 SUBSTR 在计算子串的起始位置时，一个空格占用一个字符。上述字符串"structured query language"共有 25 个字符，第 12 个字符是'q'，而起始参数 m = 12，没有指定结束字符的位置，所以默认从第 12 个字符开始到字符串的结尾。

（6）LENGTH

其函数格式为：LENGTH(column | expression)，用于计算字符串中的字符个数。实例 4-23 用于测试字符串"structured query language"中的字符数。

【实例 4-23】使用 LENGTH 函数。

```
SQL> SELECT LENGTH('structured query language')
  2  FROM dual;

LENGTH('STRUCTUREDQUERYLANGUAGE')
---------------------------------
                               25
```

（7）INSTR

其函数格式为：INSTR(column | expression, 'string', [,m], [n])，该函数的功能是在字符串 expression 中搜索字符串"string"，参数 m 和 n 用于指定搜索的开始位置和结束位置。如果没有指定 m 和 n 的值，则从字符串 expression 中搜索，如实例 4-24 所示。

【实例 4-24】使用 INSTR 函数。

```
SQL> SELECT instr('structured query language','query')
  2  FROM dual;

INSTR('STRUCTUREDQUERYLANGUAGE','QUERY')
----------------------------------------
                                      12
```

该函数返回一个数字，表明字符串'query'在字符串'structured query language'中的起始位置。

（8）LPAD|RPAD

其函数格式为：LPAD|RPAD (column | expression，n, ‘string’)，通过实例 4-25 说明该函数的作用。

【实例 4-25】使用 LPAD 函数。

```
SQL> SELECT LPAD(sal,10,'*')
  2  FROM emp
  3  WHERE sal >1500;

LPAD(SAL,10,'*')
--------------------
******1600
******2975
******2850
******2450
******5000
******3000
```

```
已选择 6 行。
```

在函数 LPAD(sal,10,'*')中，sal 是表 emp 中的列名，10 表示该函数的输出结果需要 10 个字符，而“*”号表示如果列 sal 的值不足 10 个字符，则在 sal 值的左边用“*”号补充。如果此时函数为 RPAD(sal,10,'*')，则在 sal 值的右边用“*”号补充。读者可以自己测试。函数 LPAD 和 RPAD 的作用就是在输出结果中增加一些补充信息，使得输出结果更具有可读性。

（9）TRIM

其函数格式为：TRIM (leading|trailing|both, trim_character FROM Trim_source)，该函数的作用是在字符串中剪切一个字符，输出结果是一个字符串。其中参数 leading|trailing|both 的作用分别是函数从源字符串的头部删除要剪切的字符，还是从尾部和两边删除要剪切的字符，默认是 both。

【实例 4-26】使用 TRIM 函数。

```
SQL>SELECT trim ('S' FROM 'SQL is an easy Database languageS')
  2  FROM dual;

TRIM('S'FROM'SQLISANEASYDATABAS
-------------------------------
QL is an easy Database language
```

输出结果显示已经把源字符串'S' FROM 'SQL is an easy Database languageS'中的第一个 S 和最后一个 S 都删除了。

（10）REPLACE

其函数格式为：REPLACE (text,search_string,replacement_string)，该函数把源字符串（text）中的某个字符串（search_string）替换为另一个字符串（replacement_string）。该函数很简单，给出一个实例，以供读者体会。

【实例 4-27】使用 REPLACE 函数。

```
SQL> SELECT replace ('sql is an easy Database language','sql',
'Structured Query Language')
  2  FROM dual;

REPLACE('SQLISANEASYDATABASELANG
--------------------------------
Structured Query Language is an easy Database language
```

该函数将源字符串中的 sql 替换为 Structured Query Language，用完整的英语单词更容易理解。

2. 数字型单行函数

数字型单行函数实现对数字的处理，其输出也是数字类型。它包括如下三个函数：

- ROUND(column/expression , n)。
- TRUNC(column/expression , n)。
- MOD(m , n)。

下面依次介绍这些函数的具体使用方法。

（1）ROUND 函数

该函数的作用是输出用户指定的小数位，如数字 32.1415，用户可以要求只输出小数点后的 3 位，但是该函数处理数字时使用四舍五入的规则，如实例 4-28 所示。

【实例 4-28】使用 ROUND 函数。

```
SQL> SELECT round(32.1415,3)
  2  FROM dual;

ROUND(32.1415,3)
-------------------
           3.142
```

如果该函数参数 n 为负数，则表示要求保留相应的整数位，如实例 4-29 所示。

【实例 4-29】保留整数位。

```
SQL> SELECT round(32.1415,-1)
  2  FROM dual;

ROUND(32.1415,-1)
-----------------
             30
```

（2）TRUNC 函数

该函数的作用是截断一个数字，只保留小数点后一定的位数，该函数处理数字时不使用四舍五入的规则，显然 Oracle 使用截断一词的用意也是如此。

【实例 4-30】使用 TRUNC 函数 1。

```
SQL> SELECT trunc (32.1414,3)
  2  FROM dual;

TRUNC(32.1415,3)
----------------
        32.141
```

【实例 4-31】使用 TRUNC 函数 2。

```
SQL> SELECT trunc (32.1415,-1)
  2  FROM dual;

TRUNC(32.1415,-1)
-----------------
             30
```

（3）MOD 函数

该函数的作用是求余数，如实例 4-32 所示。

【实例 4-32】使用 MOD 函数（够除）。

```
SQL> SELECT mod(1000,400)
  2  FROM dual;

MOD(1000,400)
-------------
          200
```

该实例中用 1000 除以 400，商为 2，此时余数是 200，即 2*400＋200（余数）＝1000，所以经过计算之后的结果是 200，但是如果不够除又怎么办呢？用 100 除以 400 显然不够除，下面通过实例 4-33 测试结果。

【实例 4-33】使用 MOD 函数（不够除）。

```
SQL> SELECT mod (100,400)
  2  FROM dual;

MOD(100,400)
------------
         100
```

显然 100/400 不够除，商为 0，此时余数是 100，即 0*400＋100（余数）＝100。

3. 日期型单行函数

Oracle 使用内部数字格式存储日期。默认的日期显示和输入格式为 DD-MON-RR。有效的日期从公元前 4712 年 1 月 1 日到公元 9999 年 12 月 31 日。日期在数据库中的内部存储格式为：世纪、年、月、日、时、分、秒。不论外部的日期形式如何改变，数据库对日期的内部存储格式是不会变的。

Oracle 提供了用于操作或显示日期的函数，它们包括：SYSDATE、MONTHS_BETWEEN、ADD_MONTHS、NEXT_DAY、LAST_DAY。下面依次介绍这些函数。

（1）SYSDATE 函数

该函数返回系统的当前日期，该日期受操作系统限制，即 Oracle 数据库读取操作系统的时间，如实例 4-34 所示。

【实例 4-34】查询 SYSDATE 的值。

```
SQL> SELECT sysdate
  2  FROM dual;

SYSDATE
---------
06-JUN-09
```

SYSDATE 函数也可以进行算数运算，日期函数和一个数字相加减，可以得到一个日期值，这个数字代表一个天数，如实例 4-35 所示。

【实例 4-35】包含 SYSDATE 运算的查询。

```
SQL> SELECT SYSDATE + 7 , SYSDATE - 7
```

```
  2  FROM dual;

SYSDATE+7 SYSDATE-7
--------- ---------
13-JUN-09 30-MAY-09
```

两个日期型数据相减时，会得到一个数字型数据，如实例 4-36 所示。

【实例 4-36】日期相减的运算查询。

```
SQL> SELECT to_date('06-JUN-10') - SYSDATE
  2  FROM dual;

TO_DATE('06-JUN-10')-SYSDATE
----------------------------
               364.460139
```



函数 to_date()是把字符型数据转换为日期型数据的方法。上述实例得到的数字型数据表示天数，即某个日期距离当前日期还有多少天，在上例中当前日期是 06-JUN-09，而某个日期是 06-JUN-10，二者相差几乎一年，上例也验证了这个结果。

还可以在日期型数据上加一些小时数，如在当前日期上增加 20 个小时，得到的仍然是日期型数据。但是，此时的小时数必须除以 24，如实例 4-37 所示。

【实例 4-37】在日期型数据上加小时数的查询。

```
SQL> SELECT sysdate + 20/24
  2  FROM dual;

SYSDATE+2
---------
07-JUN-09
```

此时，输出结果比当前日期多了一天，因为当前笔者的日期为 2009-6-6 13:05:51，所以加 20/24 小时后是 2009-6-7。但是如果将 20/24 改为 1/24，即只在当前日期上增加一小时，小时会改变，但是日期不会改变，如实例 4-38 所示。

【实例 4-38】在当前日期上增加一小时的查询。

```
SQL> SELECT sysdate + 1/24
  2  FROM dual;

SYSDATE+1
---------
06-JUN-09
```



为了使上述日期数据的实例运行正确，需要读者设置数据库的字符集为美国英语，可使用如下指令实现：

```
SQL>alter session set NLS_DATE_LANGUAGE = 'AMERICAN';
```

（2）MONTHS_BETWEEN(date,date)

该函数的参数为两个日期，得到两个日期之间的月数，即两个日期间相差几个月，如实例 4-39 所示。

【实例 4-39】使用 MONTHS_BETWEEN 函数。

```
SQL>SELECT months_between('06-JUN-10','06-JUN-09'),
  2  FROM dual;
MONTHS_BETWEEN('06-JUN-10','06-JUN-09')
 ----------------------------------------------------------
                              12
```

如果函数 MONTHS_BETWEEN 中第一个参数早于第二个参数，则得到一个负值，如实例 4-40 所示。

【实例 4-40】使用 MONTHS_BETWEEN 函数。

```
SQL> SELECT months between('06-JUN-08','06-JUN-09')
  2  FROM dual;

MONTHS_BETWEEN('06-JUN-08','06-JUN-09')
---------------------------------------
                             -12
```

（3）ADD_MONTHS(date,n)

该函数的参数为日期型数据和一个数字型数据 n，函数功能是把 n 个月添加到日期型数据上。输出结果仍为日期型数据，如实例 4-41 所示。

【实例 4-41】使用 ADD_MONTHS 函数。

```
SQL> SELECT add months(SYSDATE , 4)
  2  FROM dual;

ADD_MONTH
---------
06-OCT-09
```

系统的 SYSDATE 为 06-JUN-09，在这个日期上增加 4 个月就是 06-OCT-09。

（4）NEXT_DAY（date,string）

该函数的参数为一个日期型数据，输出为该日期的下一个指定的日期，如实例 4-42 所示。

【实例 4-42】使用 NEXT_DAY 函数。

```
SQL> SELECT next_day(sysdate,'Saturday')
  2  FROM dual;

NEXT_DAY(
---------
13-JUN-09
```

该实例是希望得到从当前日期开始，第一个 Saturday 的日期。当前日期是 06-JUN-09 星期六，

所以下一个星期六为 13-JUN-09。

（5）LAST_DAY（date）

该函数返回参数中日期的最后一天的日期，如实例 4-43 所示。

【实例 4-33】使用 LAST_DAY 函数。

```
SQL> SELECT last_day(sysdate)
  2  FROM dual;

LAST_DAY(
---------
30-JUN-09
```

上述实例输出本月的最后一天是几号。

4.2.4　使用空值处理函数

空值是非常特殊的值，既不能说它不存在，也不能说它是零。空值表示一类没有定义的值，具有不确定性。当然对于空值的运算也具有特殊性，因为具有不确定性的值是无法和一类具有确定性的值进行逻辑或算数运算的，所以 Oracle 提供了一类空值处理函数，通过这些函数实现空值（NULL）的运算。下面依次介绍什么是空值以及与空值相关的函数，这些函数包括 NVL 函数、NVL2 函数、NULLIF 函数、COALESCE 等。

1. 空值的定义

空值是一类没有定义的、具有不确定性的值。在数据表中，这类值无法表示，更无法显示。在 SCOTT 用户的 emp 表中有空值，如实例 4-44 所示。

【实例 4-44】查询 emp 表中的空值。

```
SQL> col empno for 9999
SQL> col sal for 9999
SQL> col comm for 9999
SQL> col mgr for 9999
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2   FROM emp
  3*  order by job

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
----- ---------- --------- ----- ---------- ----- -----
 7902 FORD       ANALYST    7566  03-12 月-81  3000
 7369 SMITH      CLERK      7902  17-12 月-80   800
 7900 JAMES      CLERK      7698  03-12 月-81   950
 7934 MILLER     CLERK      7782  23-1 月 -82  1300
 7566 JONES      MANAGER    7839  02-4 月 -81  2975
 7782 CLARK      MANAGER    7839  09-6 月 -81  2450
 7698 BLAKE      MANAGER    7839  01-5 月 -81  2850
 7839 KING       PRESIDENT        17-11 月-81  5000
```

```
  7499 ALLEN      SALESMAN   7698   20-2月 -81  1600     300
  7654 MARTIN     SALESMAN   7698  28-9月 -81  1250     1400
  7844 TURNER     SALESMAN   7698  08-9月 -81  1500       0

EMPNO ENAME      JOB        MGR HIREDATE     SAL  COMM
----- ---------- --------- ----- ---------- ----- -----
  7521 WARD       SALESMAN   7698  22-2月 -81   1250     500

已选择 12 行。
```

在上述输出中除了 SALSMAN 有 COMM 佣金外，其他职位根本没有，所以在表中相应的 COMM 列的值为空值（NULL），但是这个值在表中是没有显示的。

空值可以用于表达式运算，但是因为空值不是具体的值，具有不确定性，所以下面实例 4-45 的查询不成功。

【实例 4-45】查询表 emp 中“comm=NULL”的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
  3  WHERE comm = NULL;

未选定行
```

通过上例可以看出，空值（NULL）不是某个值，可以用 NULL 表示它，但是不能直接用于计算。

那么如何实现上例中 WHERE 子句中的条件呢，即如何判断某列的值为空值（NULL）呢？Oracle 提供了 IS NULL 和 IS NOT NULL 运算符来处理这个运算，如实例 4-46 使用的是 IS NULL。

【实例 4-46】查询表 emp 中“comm is NULL”的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
  3  WHERE comm IS NULL;

EMPNO ENAME      JOB        MGR HIREDATE     SAL  COMM
------------- ---------------- ------------------- ----- -----------------
  7369 SMITH      CLERK       7902  17-12月-80    800
  7566 JONES      MANAGER    7839   02-4月 -81    2975
  7698 BLAKE      MANAGER    7839  01-5月 -81    2850
  7782 CLARK      MANAGER    7839  09-6月 -81    2450
  7839 KING       PRESIDENT         17-11月-81    5000
  7900 JAMES      CLERK       7698  03-12月-81    950
  7902 FORD       ANALYST     7566  03-12月-81    3000
  7934 MILLER     CLERK       7782   23-1月 -82   1300

已选择 8 行。
```

实例 4-47 使用 IS NOT NULL 查询有佣金的信息，即 COMM 列不为空的数据。

【实例 4-47】查询表 emp 中 comm IS NOT NULL 的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
```

```
  3  WHERE comm IS NOT NULL;

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
------------- --------------- ------------------ ---------- -------------
 7499 ALLEN      SALESMAN   7698   20-2 月 -81      1600   300
 7521 WARD       SALESMAN   7698   22-2 月 -81      1250   500
 7654 MARTIN     SALESMAN   7698   28-9 月 -81      1250  1400
 7844 TURNER     SALESMAN   7698   08-9 月 -81      1500     0
```

2. NVL 函数和 NVL2 函数

NVL 函数使得空值可以进行运算，它是空值转换函数。如果不使用空值转换函数，空值是无法进行运算的。NVL 函数的语法格式如下。

```
NVL(expr1,expr2)
```

其计算规则是如果 expr1 的值为空值（NULL），则返回 expr2 的值，否则返回 expr1 的值。其中表达式 expr1 和 expr2 的数据类型必须相同，它们可以是数字类型、字符类型和日期类型。下面利用实例 4-48 使用 NVL 函数计算 sal + comm 的值。

【实例 4-48】使用 NVL 函数计算 sal + comm 的值的查询。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal+NVL(comm,0)
  2  FROM emp;

EMPNO ENAME      JOB       MGR  HIREDATE   SAL+NVL(COMM,0)
----- ---------- --------- ----- ---------- ---------------
 7369 SMITH      CLERK       7902   17-12 月-80           800
 7499 ALLEN      SALESMAN   7698  20-2 月 -81          1900
 7521 WARD       SALESMAN   7698  22-2 月 -81          1750
 7566 JONES      MANAGER     7839  02-4 月 -81          2975
 7654 MARTIN     SALESMAN   7698  28-9 月 -81          2650
 7698 BLAKE      MANAGER    7839  01-5 月 -81          2850
 7782 CLARK      MANAGER    7839  09-6 月 -81          2450
 7839 KING       PRESIDENT         17-11 月-81           5000
 7844 TURNER     SALESMAN   7698  08-9 月 -81          1500
 7900 JAMES      CLERK       7698  03-12 月-81            950
 7902 FORD       ANALYST     7566   03-12 月-81           3000

EMPNO ENAME      JOB       MGR HIREDATE   SAL+NVL(COMM,0)
----- ---------- --------- ----- ---------- ---------------
 7934 MILLER     CLERK      7782    23-1 月 -82           1300

已选择 12 行。
```

从上例输出结果可以看出，COMM 列为空值的值转换为 0。如果不使用 NVL 函数进行空值转换，则无法实现计算，所以不会输出任何结果，如实例 4-49 所示。

【实例 4-49】不使用 NVL 函数计算 sal + comm 的值的查询。

```
SQL> SELECT ename,sal,comm,sal+comm
  2  FROM emp
```

```
  3  order by sal+comm;

ENAME         SAL  COMM   SAL+COMM
---------- ----- ----- ----------
TURNER       1500     0       1500
WARD         1250   500       1750
ALLEN        1600   300       1900
MARTIN       1250  1400       2650
SMITH         800
JAMES         950
MILLER       1300
FORD         3000
JONES        2975
BLAKE        2850
CLARK        2450

ENAME         SAL  COMM   SAL+COMM
---------- ----- ----- ----------
KING         5000

已选择 12 行。
```

显然，上例输出中由于 sal+comm 计算时， comm 列有的值为空值，所以无法计算，自然也无法显示这样的计算结果。

NVL2 函数是对 NVL 函数功能的增强。NVL2 函数的格式为：

```
NVL2(expr1,expr2,expr3)
```

其基本功能就是实现空值（NULL）的转换。其计算规则是：如果 expr1 为空，则返回表达式 expr3 的值，如果 expr1 不为空，则返回表达式 expr2 的值。其中 expr1 为任何数据类型，而表达式 expr2 和 expr3 为除 LONG 数据类型以外的任何数据类型。

【实例 4-50】使用 NVL2 实现包含 sal + comm 的值的查询。

```
SQL> SELECT ename,nvl2(comm,sal+comm,sal)
  2  FROM emp;

ENAME      NVL2(COMM,SAL+COMM,SAL)
---------- -----------------------
SMITH                          800
ALLEN                         1900
WARD                          1750
JONES                         2975
MARTIN                        2650
BLAKE                         2850
CLARK                         2450
KING                          5000
TURNER                        1500
JAMES                          950
FORD                          3000
```

```
ENAME      NVL2(COMM,SAL+COMM,SAL)
---------- -----------------------
MILLER                        1300

已选择 12 行。
```

上例中 NVL2(comm,sal+comm,sal)的计算规则是如果 comm 的值为空，则返回 sal 的值，如果 comm 的值不为空，则返回 sal+comm 的值。

3. NULLIF 函数

NULLIF 函数用于比较两个表达式，如果二者相等则返回空值 NULL，如果二者不等则返回第一个表达式的值。要求第一个表达式的值不能为 NULL。其语法格式为：

```
NULLIF(expr1,expr2)
```

【实例 4-51】使用 NULLIF 函数。

```
SQL> select ename,length(ename) "expr1",job,length(job) "expr2",
  2  NULLIF(length(ename),length(job)) "Comparision Result"
  3  FROM EMP;

ENAME          expr1  JOB            expr2     Comparision Result
------------------------ ---------- ---------------------------
SMITH              5  CLERK            5
ALLEN              5  SALESMAN         8                5
WARD              4  SALESMAN         8                4
JONES              5  MANAGER          7                5
MARTIN             6  SALESMAN         8                6
BLAKE              5  MANAGER          7                5
CLARK              5  MANAGER          7                5
KING               4  PRESIDENT          9                4
TURNER             6  SALESMAN         8                6
JAMES              5  CLERK            5
FORD               4  ANALYST          7                4

ENAME          expr1  JOB            expr2      Comparision Result
------------------------ ---------- --------------------------- -
MILLER             6 CLERK            5                6

已选择 12 行。
```

在上例中函数 NULLIF 中有两个表达式，一个是 length(ename)，另一个是 length(job)，函数首先比较这两个表达式的值，二者相等则返回空值 NULL，如上例中的第一行记录，如果二者值不等，则返回第一个表达式的值。

4. COALESCE 函数

COALESCE 函数返回该函数中第一个不为 NULL 的表达式的值。其语法格式为：

```
COALESCE(expr1,expr2,…exprn)
```

下面用实例 4-52 说明如何使用该函数。

【实例 4-52】使用 COALESCE 函数。

```
SQL> SELECT ename "Employ name",job,COALESCE(comm,1) "Comm"
  2  FROM emp
  3* ORDER BY job

Employ_nam JOB          Comm
---------------- -------------------------- ----------
FORD       ANALYST          1
SMITH      CLERK             1
JAMES      CLERK             1
MILLER     CLERK             1
JONES      MANAGER          1
CLARK      MANAGER         1
BLAKE      MANAGER         1
KING       PRESIDENT         1
ALLEN      SALESMAN       300
MARTIN     SALESMAN      1400
TURNER     SALESMAN         0

Employ nam JOB          Comm
------------------------------------------ ----------------
WARD       SALESMAN       500

已选择 12 行。
```

在上例中，我们使用函数 COALESCE 查询雇员（employee）的佣金，如果没有佣金则显示 1，如果佣金值不为空 NULL，则返回佣金值。上例中 JOB 为 SALESMAN 的雇员都有佣金，而其他雇员没有佣金，因为这些雇员的 COMM 列的值为 NULL。

4.2.5 使用逻辑判断功能

在高级程序设计语言中，为语句的逻辑结构设计了逻辑判断语句，如 IF-THEN-ELSE。在 SQL 语句中 Oracle 也提供了两个函数来实现逻辑判断的功能，则这两个函数分别是 CASE 表达式和 DECODE 函数。

1. CASE 表达式

CASE 表达式用于逻辑判断，为了说明其用法，先给出其语法结构，如下所示：

```
CASE expr WHEN comparison_expr1 THEN return_expr1
        [WHEN comparison expr2] THEN return expr2
        WHEN comparison_exprn] THEN return_exprn
        ELSE else_expr]
END
```

该表达式首先比较 expr 和 comparison_expr1，如果二者相等，则返回 return_expr1，否则比较

expr 和 comparison_expr2，如果二者相等则返回 return_expr2，否则继续判断，如果都不满足，最后返回 ELSE 后的 else_expr。下面通过实例 4-53 说明如何使用该表达式。

【实例 4-53】使用 CASE 表达式。

```
SQL>  SELECT ename,job,sal,
  2   CASE job WHEN 'SALESMAN' THEN 1.20*sal
  3            WHEN 'MANAGER' THEN 1.30*sal
  4            WHEN 'ANALYST' THEN 1.40*sal
  5  ELSE sal END "Last Salary"
  6  FROM emp
  7 ORDER BY job;

ENAME     JOB           SAL Last Salary
-------------------- ------------------------- ---------- -----------
FORD      ANALYST       3000      4200
SMITH     CLERK          800       800
JAMES     CLERK          950       950
MILLER    CLERK         1300      1300
JONES     MANAGER       2975      3867.5
CLARK     MANAGER       2450      3185
BLAKE     MANAGER       2850      3705
KING      PRESIDENT     5000        5000
ALLEN     SALESMAN      1600      1920
MARTIN    SALESMAN      1250      1500
TURNER    SALESMAN      1500      1800

ENAME     JOB           SAL Last Salary
-------------------- ------------------------- ------------- -----------
WARD      SALESMAN      1250      1500

已选择 12 行。
```

在该实例中，对岗位为 SALESMAN、MANAGER 和 ANALYST 的雇员进行适当加薪，通过实例可以看到，这些员工的工资得到增加，通过前后对比可以很明显看出这个变化。



说 明

在表达式 CASE 中，表达式 expr、comparison_exprn 和 return_expr 必须是相同的数据类型，这些数据类型是 CHAR、VARCHAR2、NCHAR 或者 NVARCHAR2。

2. DECODE 函数

DECODE 函数同 CASE 表达式具有相同的功能，不过 DECODE 函数使用更简单，其语法格式为：

```
DECODE(col|expression, search1,result1
                   [,search2, result2,…,]
                   [,default])
```

该函数的执行过程是首先判断 search1 的值是否和 col 或 expression 的值相等，如果相等，则返

回 result1，否则判断 search2 的值是否和 col 或 expression 的值相等，如果相等则返回 result2 的值，依次判断，如果都不相等，则返回默认值 default。下面通过实例 4-54 说明如何使用 DECODE 函数。

【实例 4-54】使用 DECODE 函数。

```
SQL> SELECT ename,job,sal,
  2  DECODE(job,'SALESMAN',1.20*sal,
  3            'MANAGER',1.30*sal,
  4            'ANALYST',1.40*sal,
  5         sal)
  6  Last_Salary
  7  FROM emp
  8  ORDER BY job;

ENAME      JOB              SAL  LAST_SALARY
---------- --------- ---------- -----------
FORD       ANALYST         3000        4200
SMITH      CLERK            800         800
JAMES      CLERK            950         950
MILLER     CLERK           1300        1300
JONES      MANAGER         2975      3867.5
CLARK      MANAGER         2450        3185
BLAKE      MANAGER         2850        3705
KING       PRESIDENT       5000        5000
ALLEN      SALESMAN        1600        1920
MARTIN     SALESMAN        1250        1500
TURNER     SALESMAN        1500        1800

ENAME      JOB              SAL  LAST_SALARY
---------- --------- ---------- -----------
WARD       SALESMAN        1250        1500

已选择 12 行。
```

上例的输出结果和 CASE 表达式中实例的输出结果完全一样。函数判断过程同 CASE 表达式的判断过程一样。

4.2.6 使用分组函数

分组函数对表中的多行进行操作，而每组返回一个计算结果。常用的分组函数包括以下几个。

- AVG。
- SUM。
- MAX。
- MIN。
- COUNT。

这些函数的统一用法如下所示。

```
SELECT  [column,] group_function_name(column),…
FROM    tablename
[WHERE   condition]
[GROUP BY column]
[ORDER BY column];
```

1. AVG 和 SUM 函数

实例 4-55 为使用 AVG 函数和 SUM 函数查询 SCOTT 用户中表 emp 内员工的平均工资和所有员工的工资总和。

【实例 4-55】使用 AVG 函数和 SUM 函数。

```
SQL> SELECT AVG(sal) "平均工资",SUM(sal) "总工资"
  2  FROM emp;

  平均工资     总工资
---------- ----------
2077.08333      24925
```

该实例计算了表 emp 中所有员工的平均工资和总工资。

2. MAX 和 MIN 函数

与 AVG 和 SUM 函数不同，MAX 和 MIN 函数既可以操作数字型数据，也可以操作字符型和日期型数据，如实例 4-56 所示，用于计算表 emp 中员工的最高工资和最低工资。

【实例 4-56】使用 MAX 和 MIN 函数。

```
SQL> SELECT MAX(SAL) "Highest salary", MIN(SAL) "Lowest salary"
  2  FROM emp;

Highest salary Lowest salary
-------------- -------------
          5000           800
```

【实例 4-57】计算最早雇佣员工的日期和最晚雇佣员工的日期。

```
SQL> SELECT MAX(HIREDATE) "Last day", MIN(HIREDATE) "First day"
  2  FROM EMP;

Last day   First day
---------- ----------
23-1月 -82 17-12月-80
```

3. COUNT 函数

该函数返回经计算得到的返回的行数，包括空行和重复的行，如实例 4-58 所示为查询表 emp 中所有的记录个数，即表中行数。

【实例 4-58】使用 COUNT()函数。

```
SQL> SELECT count(*) "表中的行数"
  2  FROM emp;
```

```
表中的行数
----------
       12
```

实例 4-59 使用关键字 DISTINCT 返回不同的 JOB 类型数量，即去掉重复的 JOB 行记录。

【实例 4-59】使用包含 DISTINCT 的 COUNT 函数。

```
SQL> SELECT count(distinct job)
  2  FROM emp;

COUNT(DISTINCTJOB)
------------------
               5
```

从实例 4-58 和实例 4-59 可以看出，该表中共有 12 行记录，共有 5 种工作职位。

4. GROUP BY 子句

在前面的内容中，使用 AVG 和 SUM 函数查询了表 emp 中的员工平均工资和总工资数，但是如果要查询每个工作职位的员工平均工资和总工资之和又该如何计算呢？此时需要使用 GROUP BY 子句，按照工作职位分组，然后再计算，如实例 4-60 所示。

【实例 4-60】使用 GROUP BY 函数。

```
SQL>SELECT JOB, AVG(sal) "平均工资",SUM(sal) "总工资"
  2  FROM emp
  3* GROUP BY JOB

JOB          平均工资      总工资
--------- ---------- ----------
ANALYST         3000       3000
CLERK     1016.66667       3050
MANAGER   2758.33333       8275
PRESIDENT       5000       5000
SALESMAN        1400       5600
```

在上述查询结果中，是按照职位名字的顺序排序的，如果想按照总工资数的多少顺序排列，则需要使用 ORDER BY 子句，如实例 4-61 所示。

【实例 4-61】使用 ORDER BY 子句。

```
SQL> SELECT JOB, AVG(sal) "平均工资",SUM(sal) "总工资
  2  FROM emp
  3  GROUP BY JOB
  4  ORDER BY "总工资";

JOB          平均工资      总工资
--------- ---------- ----------
ANALYST         3000       3000
CLERK     1016.66667       3050
```

```
PRESIDENT       5000       5000
SALESMAN        1400       5600
MANAGER   2758.33333        8275
```

显然，这样的结果可以一目了然总工资的排序，在函数 AVG 和 SUM 后只用别名，也使得输出结果更加容易阅读。

分组函数可以嵌套使用，如实例 4-62 所示，计算按照工作职位分类最高平均工资和最低平均工资数。

【实例 4-62】使用分组嵌套函数。

```
SQL> SELECT MAX(AVG(sal)) ,MIN(AVG(sal))
  2  FROM EMP
  3  GROUP BY JOB;

MAX(AVG(SAL)) MIN(AVG(SAL))
------------- -------------
         5000    1016.66667
```

在执行实例 4-62 中的语句时，Oracle 首先会实现全表扫描，按照 JOB 分类，再计算每类的平均值，最后计算这些平均值的最大值和最小值。

5. HAVING 子句

在分组函数中，不能使用 WHERE 子句限制分组函数，所以 Oracle 设计了 HAVING 子句来执行对分组函数的某些限制，如实例 4-63 所示使用 HAVING 子句限制了 AVG(sal)>2000，即只显示平均工资大于 2000 的职位信息。

【实例 4-63】使用 HAVING 子句。

```
SQL> SELECT job,AVG(sal)
  2  FROM emp
  3  HAVING AVG(sal)>2000
  4  GROUP BY job;

JOB          AVG(SAL)
--------- ----------
ANALYST         3000
MANAGER   2758.33333
PRESIDENT       5000
```

该实例中也可以使用 ORDRE BY 子句对 AVG(sal)进行排序，使得输出更容易阅读。

【实例 4-64】在分组函数中使用 ORDER BY 子句。

```
SQL>  SELECT job,AVG(sal)
  2   FROM emp
  3   HAVING AVG(sal)>2000
  4   GROUP BY job
  5  ORDER BY 2;
```

```
JOB         AVG(SAL)
--------- ----------
MANAGER   2758.33333
ANALYST         3000
PRESIDENT       5000
```


说 明

ORDER BY 2 和 ORDER BY AVG(SAL) 的效果一样，只是为了书写方便，使用数字 2 表示按照第二列排序。

4.3 数据操纵语句（DML）

数据操纵语句（Data Manipulation Language）实现对表中数据的各种操作，如向表中插入数据、删除一行数据或者更新表中的行数据。无论读者使用何种高级语言开发连接数据库的程序，数据操作语句都是使用频率最高的。下面依次介绍 INSERT 语句、UPDATE 语句和 DELETE 语句。

4.3.1 INSERT 语句

利用 INSERT 语句向表中添加一行数据的语法格式如下。

```
INSERT INTO tablename [(column [,column …] ) ]
VALUES     (value [, value… ] )
```

在上述语法格式中“()”中的“[]”号表示可选部分，即可向表中一列或多列插入数据。VALUES 后是插入数据的值，这些值和 tablename 后的列名一一对应。

- tablename 是要插入数据的表名字，要求用户对该表有操作权限。
- column 是该表中的列名，用户需要向这些列插入数据，可以是一列，也可以是多列，如果向表中的所有列插入一行数据，也可以不使用任何 column，但是需要用户清楚知道该表中的列名和列的属性。
- values 是要插入的和列对应的值，插入值的数据类型必须和 column 的数据类型相匹配。

下面需要向 SCOTT 用户的 dept 表中添加一行数据，即增加一行记录，其中 DEPTNO 为 50，DNAME 为 MARKETING，LOC 为 NEW YORK。为了验证添加结果，我们先查询一下表中的数据。

【实例 4-65】查询表中的所有数据。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

查看一下 dept 中列的数据类型。

```
SQL> desc dept;
 名称                    空?      类型
 ----------------------------------------------------- --------
 DEPTNO           NOT NULL  NUMBER(2)
 DNAME                     VARCHAR2(14)
 LOC                        VARCHAR2(13)
```

其中 DEPTNO 为数字型，不允许为空（NOT NULL），DNAME 和 LOC 都为变长字符型。在插入数据时，与字符型列相对应的值用英文输入法的单引号' '括起来。

输出结果表明当前表中有 4 行数据，我们再向表中添加一行数据。

【实例 4-66】向表 dept 中插入一行数据。

```
SQL> INSERT INTO dept (deptno,dname,loc)
  2  VALUES          (50,'MARKETING','NEW YORK');

已创建 1 行。
```

输入提示已经成功创建一行，下面为了验证 INSERT 语句的执行结果，再次查询表 dept 中的数据。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
```

显然，输出结果显示已经成功添加了一行数据。

如果需要向 DEPT 表中添加一行数据，而只有部门号 DEPTNO 和部门名称 DNAME，但是地点 LOC 还没有确定，可以在 tablename 后的列中只包含 DEPTNO 和 DNAME 两列，如实例 4-67 所示。

【实例 4-67】向表 dept 中插入一行数据（没有 LOC 列对应的值）。

```
SQL> INSERT INTO dept(deptno,dname)
  2  VALUES          (60,'ACCOUNTING');

已创建 1 行。
```

再用实例 4-68 验证插入结果。

【实例 4-68】查询插入结果。

```
SQL> SELECT *
  2  FROM dept;
```

```
   DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
        60 ACCOUNTING

已选择 6 行。
```

在当前表中新增了一行记录，其中部门号 DEPTNO 为 60，部门名字为 ACCOUNTING，而部门所在地没有值，Oracle 使用空值（null）填充，此时读者也可以使用实例 4-69 得到同样的插入效果。

【实例 4-69】向表中插入一行数据。

```
SQL> INSERT INTO dept (deptno,dname,loc)
  2  VALUES          (60,'ACCOUNTING',NULL);

已创建 1 行。
```

还有一种插入方式，即从另一张表拷贝数据，它涉及两张表，这种方式的语法结构如下：

```
INSERT INTO tablename [(column [,column …] ) ]
SELECT column [,column…]
FROM   another tablename
WHERE clause
```

4.3.2 UPDATE 语句

UPDATE 语句用于更新表中的数据，如在表 dept 中，需要把刚插入记录的 LOC 部门地点设置为 NEW YORK。此时就需要利用 UPDATE 语句来更新表中的该行记录，语法格式如下：

```
UPDATE   tablename
SET      column = value [, column = value, … ]
[WHERE   condition];
```

语法解释如下。

- tablename：要更新的表名。
- column：要更新的列。
- value：是要更新的列的值。
- condition：通过条件限制要更新的列所在的行。

在实例 4-67 中在表 dept 中新增了一行记录，DEPTNO 为 60，DNAME 为‘ACCOUNTING’，但是没有确定 LOC 部门所在地。把部门所在地设置为 NEW YORK，并用实例 4-70 说明如何使用 UPDATE 语句。

【实例 4-70】使用 UPDATE 语句更新表 DEPT 中的数据。

```
SQL> UPDATE dept
  2  SET  LOC ='NEW YORK'
  3  WHERE DEPTNO = 60;

已更新 1 行。
```

查询更新结果，如实例 4-71 所示。

【实例 4-71】查询实例 4-70 的更新结果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
        60 ACCOUNTING     NEW YORK

已选择 6 行。
```

查询结果显示更新结果正确，在实例 4-69 中 LOC 的值是直接给出的，Oracle 也允许使用一个子查询，赋予 LOC 值，语法格式如下：

```
UPDATE table
SET      column =
                (SELECT column
                 FROM tablename
                 WHERE conditon)
        [,
column =
                 (SELECT column
                 FROM tablename
                 WHERE conditon)]
[WHERE condition];
```

关于在 UPDATE 语句中使用子查询，这里不再详细介绍，读者可以自己去尝试一下。

4.3.3　DELETE 语句

DELETE 语句用于删除不需要的记录，该语句使用简单，语法格式如下。

```
DELETE [FROM]  tablename
[WHERE          condition];
```

语法解释如下。

- tablename：要删除的数据所在的表名。
- condition：限制要删除的行，该条件可以是指定具体的列名、表达式、子查询或者比较运算符。

【实例 4-72】删除表 dept 中 DEPTNO 为 60 的记录。

```
SQL> DELETE FROM dept
  2  WHERE DEPTNO = 60;

已删除 1 行。
```


说 明

在 DELETE 语句中的 FROM 关键字是可选的，使用 FROM 关键字更合乎英语的语法习惯，容易记忆。WHERE 子句也是可选的。如果不使用 WHERE 子句，将删除表中的所有行。

4.4 本章小结

本章对 SQL 语句进行了概述。通过 SQL 语句中的简单查询语句，使得读者对 SQL 语句、算数运算、别名及 DISTINCT 运算有直观的认识。在书写 SQL 语句时，要注意书写规范。函数增强了 SQL 语句的功能，使得大量的运算得以简化，本章简单介绍了单行函数和分组函数，熟练使用这些函数对于读者使用 SQL 语句很有帮助。数据操作语句也是 SQL 语句中经常使用的，如插入数据 INSERT、更新数据 UPDATE、删除数据 DELETE 等。本章还着重介绍了空值 NULL 的概念，及如何操作空值的计算。读者需要很好地理解空值 NULL，并掌握空值的相关运算。

第 5 章

◀ 创建Oracle数据库 ▶

本章将讲解如何创建 Oracle 数据库，首先需要说明创建数据库不是 DBA 的主要工作，但是理解如何创建数据库也是 DBA 需要理解的内容，本章将通过图形化工具 DBCA 创建数据库、在安装数据库软件时创建数据库以及手工创建数据库三种方式，讲解如何创建 Oracle 数据库。

5.1　创建数据库的前提条件

要创建 Oracle 数据库，首先需要获得 SYSDBA 系统权限，在讲解系统权限时我们提到了 SYSDBA 用户，其实它是一个角色，是一些高级权限的集合，如创建数据库、关闭数据库等。

其次是内存大小是否满足 Oracle 数据库实例 SGA 的要求，实例的启动涉及到一些进程的运行和数据库内存的分配，如果内存不足会造成虚拟内存的使用，更严重的是内存不足造成一些进程无法顺利运行，根本无法启动数据库。

最后就是根据业务需求对磁盘空间需求做出评估，如对数据文件、控制文件和重做日志文件的大小评估与磁盘分配。

对数据库各种文件进行部署规划时，对于存在竞争的数据文件要放在不同的磁盘上，以免 I/O 竞争（操作系统的 I/O 是耗时较长的行为），如重做日志文件和归档日志文件就不应该放在一个磁盘上。而对于控制文件要进行多路复用，Oracle 要求将多个（一般是三个，也是默认值）控制文件部署在不同的磁盘设备上，在数据库结构发生变化后，如创建了表空间要及时备份控制文件，对于重做日志文件同样需要多路复用，每个重做日志组中至少有两个数据成员，而这些日志组成员最好分布在不同的磁盘上，以减少磁盘损坏造成重做日志都无法使用。

对于数据文件要求其命名要易于维护。为了最小化磁盘碎片，把具有不同声明周期（指被创建和回收之间的时间间隔）的数据库对象放在不同的磁盘上，如临时数据文件和临时表空间的文件（临时表空间用来排序）、减少 I/O 竞争将具有磁盘 I/O 竞争的数据库对象放在不同的表空间，如一个大表和基于该表的索引要分开放置，而且这些表空间的数据文件要放在不同的磁盘上。

创建数据库有三种方式。

- 使用 DBCA（数据库配置助手）。

- 使用 CREATE DATABASE 指令。
- 在安装数据库软件时创建数据库。

DBCA（DataBase Configuration Assistant）是数据库配置助手的意思，使用它可以创建一个新的数据库，它基于 Java 的图形用户界面，在安装了 Java 虚拟机的任何操作系统平台上都可以启动，DBCA 随 Oracle Universal Installer 一同安装，也可以单独使用。选择使用 CREATE DATABASE 手工创建数据库比较麻烦，但是通过手工创建数据库读者可以更多地理解数据库的创建细节，把 DBCA 创建数据库的自动化过程分解为详细的手工操作，对于学习 Oracle 数据库是有好处的，如果是创建生产数据库最好使用 DBCA 或在安装数据库软件时就创建数据库，毕竟这种方法效率高，不容易出错，如果是出于学习的目的，读者可以手工创建数据库，一旦了解了创建数据库的底层操作，对于图形界面方式的建库过程就驾轻就熟了。

5.2　使用 DBCA 创建数据库

本节将讲解如何使用 DBCA 来协助创建数据库，相对于原始的手工方式创建数据库而言，使用 DBCA 可以明显减少建库时间，以及提高建库的效率。为了使读者从本质上理解 DBCA 建库的过程，我们对使用 DBCA 建库时生成的建库脚本文件进行分析，使大家不但能使用 DBCA，还能从根本上理解 DBCA 是如何创建数据库的，其实 Oracle 的自动化工具就是在后台运行以前手工建库的过程，只是做了一些封装使得读者少犯错，并且提高建库效率。

5.2.1　DBCA 概述

DBCA（DataBase Configuration Assistant）是 Oracle 设计的基于 Java 的图形用户界面工具，所以它可以运行在任何安装了 Java 虚拟机的操作系统平台上，在安装 Oracle 数据库时，会自动安装 Java 虚拟机，在启动 DBCA 时 Oracle 会启动虚拟机来运行 DBCA。

启动 DBCA 有两种方式，一是通过 Oracle 数据库软件安装程序的“配置和移植工具”→“Database Configuration Assistant”来启动数据库，如图 5-1 所示。

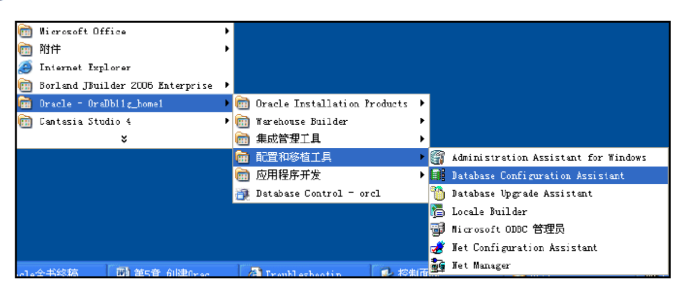


图 5-1　使用安装程序启动 DBCA

或者通过 DOS 窗口中输入 DBCA 命令来启动 DBCA，如图 5-2 所示。

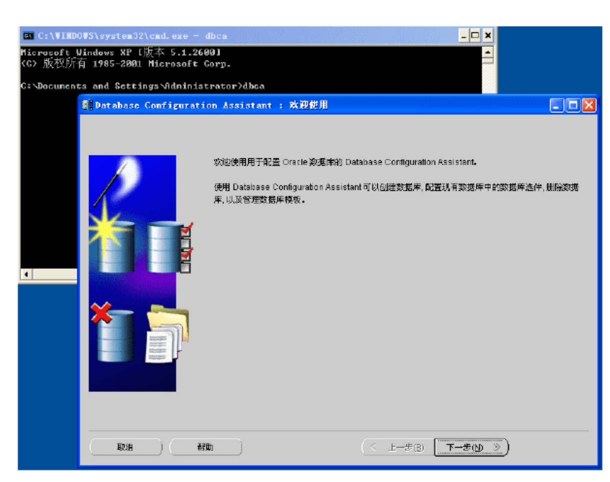


图 5-2　在 DOS 中启动 DBCA

此时，DOS 运行了 DBCA 程序，DOS 界面在 DBCA 运行期间会一直存在。

使用 DBCA 不仅能创建数据库，作为数据库配置助手它还有其他功能，即“配置数据库”、“删除数据库”、“管理创建数据库模板”等（包括使用以前的模板创建新模板、从已存在的数据库创建新模板、删除模板等操作）。

5.2.2　创建数据库的过程

创建数据库的过程如下：

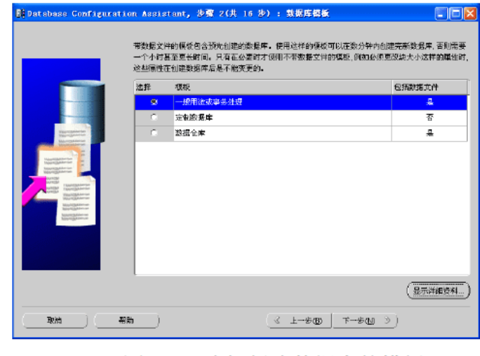
01 启动 DBCA 工具后，单击“下一步”按钮，如图 5-3 所示。



图 5-3　选择 DBCA 的操作

02 此时，对话框提示用户可以执行的操作，它也说明了 DBCA 的功能，使用 DBCA 可以“创建数据库”、“配置数据库选项”、“删除数据库”、“管理模板”以及“配置自动存储管理”。此时，当然选择创建数据库，然后单击“下一步”按钮。此时，弹出数据库模板对话框，如图 5-4 所示。

03 在图 5-4 中有三个选项来创建数据库，即“一般用途或事务处理”、“定制数据库”和“数据仓库”，根据数据库的业务要求选择其一，笔者选择“一般用途或事务处理”，单击“下一步”按钮，弹出设置数据库标识对话框，如图 5-5 所示。



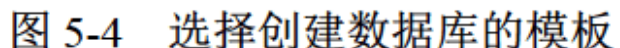


图 5-4　选择创建数据库的模板



图 5-5　设置数据库标识

04 图 5-5 中的参数“全局数据库名”提示设置数据库标识，用于标识一个唯一的全局数据库，因为数据库至少需要一个实例来维护和使用，所以还要设置实例名，一般实例名和全局数据库名相同（便于维护），在读者输入“全局数据库名”时，在实例名“SID”中会自动输入相同的内容。设置其名称为 lin，然后单击“下一步”按钮，弹出设置管理选项的对话框，如图 5-6 所示。

05 在图 5-6 中选择管理数据库的工具，默认选择使用 Enterprise Manager 来配置数据库、使用 Enterprise Manager 的 Database Control 来管理数据库，单击“下一步”按钮，弹出设置数据库口令对话框，如图 5-7 所示。此时要求设置数据库口令，这里有两种选择，一是对所有用户使用一个口令，二是“使用不同的管理口令”使得不同的用户可以设置自己的管理口令，这里选中“所有账户使用同一管理口令”单选按钮来设置统一口令。



图 5-6　设置管理选项



图 5-7　设置用户密码

06 单击图 5-7 中的“下一步”按钮，弹出网络设置的对话框，如图 5-8 所示。需要将刚创建的数据库注册到系统已有的监听程序，因为已经在安装数据库软件时创建了一个数据库，所以已存在一个监听程序。此时，选中“将此数据库注册到所有监听程序”单选按钮，单击“下一步”按钮，弹出如图 5-9 所示的存储管理对话框。

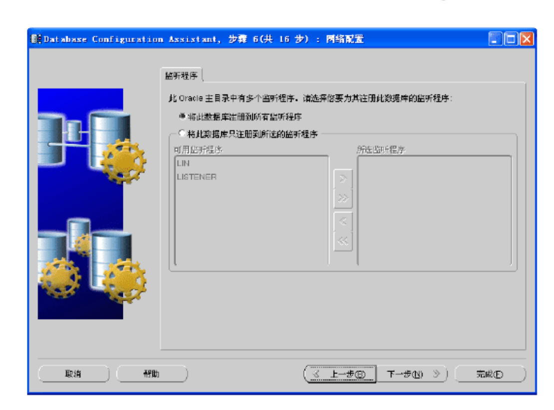
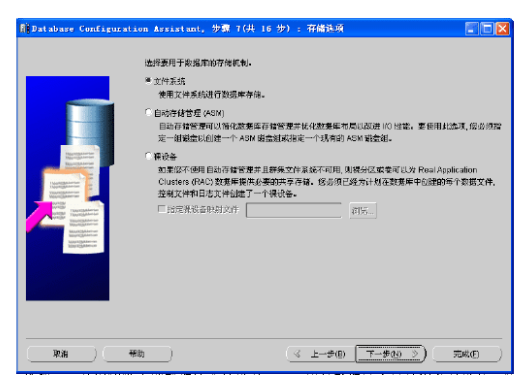


图 5-8　网络设置

07 在图 5-9 中，要求选择存储设置，此时有三个选项：文件系统存储数据库方式基于操作系统的数据库文件；自动存储管理（ASM）可以有效改进系统 I/O，但是对 ASM 系统需要配置一个磁盘组；还可以使用裸设备。此时选中“文件系统”单选按钮。单击“下一步”按钮，弹出数据库文件位置对话框，如图 5-10 所示。



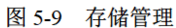


图 5-9　存储管理



图 5-10　数据库文件所在位置

08 在图 5-10 中指定数据库文件的位置，即将数据库的各种数据文件存放的目录，此时有三个选项：“使用模板中的数据库文件位置”、“所有数据库文件使用公共位置”和“使用 Oracle-Managed Files”，默认设置为“使用模板中的数据库文件位置”。

09 默认设置的 ORACLE_HOME 为 F:\app\Administrator\product\11.1.0\db_1，单击图 5-10 中的“下一步”按钮，弹出“恢复设置”对话框，如图 5-11 所示。

10 在图 5-11 中要求设置快速恢复区，在 Oracle10g 中提供了一个快闪恢复区，即用户数据库自动恢复时使用的备份数据空间，当使用 RMAN 备份时可以选择使用该恢复区，使得数据库的各类恢复行为自动化进行，该区域默认为{ORACLE_BASE}\flash_recovery_area，区域空间大小为 2G，这些参数都可以调节，快闪恢复区的存储目录通过“浏览”按钮选择，而快闪恢复区大小通过输入数值设置。单击“下一步”按钮弹出如图 5-12 所示的对话框，要求设置数据库内容。



图 5-11 恢复设置



图 5-12 数据库内容设置

11 在图 5-12 中提示是否使用“示例方案”，该方案会创建一个 EXAMPLE 的表空间，提供了如“人力资源”、“产品介质”等具体方案，主要用来演示，我们选择默认设置，即不使用“示例方案”，也不使用“定制脚本”，单击“下一步”按钮，弹出如图 5-13 所示的对话框，该对话框要求设置数据库的初始化参数。

12 这里需要设置 4 项内容，分别是“内存”、“调整大小”、“字符集”和“连接模式”。首先设置内存大小，如图 5-13 所示。Oracle 的 DBCA 自动获取了当前数据库的内存大小，根据经验设置内存大小的 40%给该数据库使用，采用默认值并选中“使用自动内存管理”复选框，内存参数在建库后可以动态修改，然后单击“调整大小”标签，如图 5-14 所示。



图 5-13 “内存”选项卡



图 5-14 “调整大小”选项卡

13 在图 5-14 中，首先需要设置 Oracle 数据库的块大小，该块是 Oracle 分配存储空间的最小单位，也是 Oracle 数据输入输出的最小单位，该参数一旦确定，将无法修改，除非重建数据库，接着是设置和该数据库连接的操作系统进程数量，该值最小为 6，因为 5 个是启动实例时必须的后台进程，另一个为用户进程。单击“字符集”标签，弹出设置字符集对话框，如图 5-15 所示。

14 在图 5-15 中设置数据库字符集，默认是采用操作系统的语言而设置，如果是中文 Windows 系统，则默认为 ZHS16GBK。单击“连接模式”标签，弹出“连接模式”选项卡，如图 5-16 所示。连接模式分为专有连接和共享连接两种模式，专有模式是指每个客户连接均启动一个专有的 PGA 与专有的相关资源。而共享模式是指多个客户连接数据库服务器时使用共享的资源空

间。我们选择默认设置，即专用服务器模式。



图 5-15　“字符集”选项卡



图 5-16　“连接模式”选项卡

15 单击图 5-16 中的“下一步”按钮，弹出如图 5-17 所示的对话框进行“安全设置”。此时，选择“保留增强的 11g 默认安全设置”单选按钮，单击“下一步”按钮，弹出如图 5-18 所示的“自动维护任务”对话框。



图 5-17　创建数据库的安全设置



图 5-18　启动数据库的自动维护任务

16 在图 5-18 中选中“启用自动维护任务”复选框，使得 Oracle 11g 数据库完成自动管理维护任务，这些任务可以在预定义的维护时间窗口内完成，系统同时限定了使用 OS 的 CPU 时间，单击“下一步”按钮，弹出如图 5-19 所示的“数据库存储”对话框。

17 在图 5-19 中，需要用户进行控制文件、表空间、数据文件以及重做日志组的参数设置，如控制文件中最大重做日志文件数、最大数据文件数等，还可以执行修改表空间的大小、创建重做日志组等操作，单击“下一步”按钮，弹出设置数据库存储参数的对话框，如图 5-20 所示。

18 图 5-20 中的创建选项有：“创建数据库”、“另存为数据库模板”和“生成数据库创建脚本”，我们选中“创建数据库”和“生成数据库创建脚本”复选框，此时在“目标目录”中自动显示脚本文件的存储目录，该目录也可以通过“浏览”按钮自行选择一个，然后单击“完成”按钮，则创建数据库，并生成数据库模板和数据库创建脚本，如图 5-21 所示。

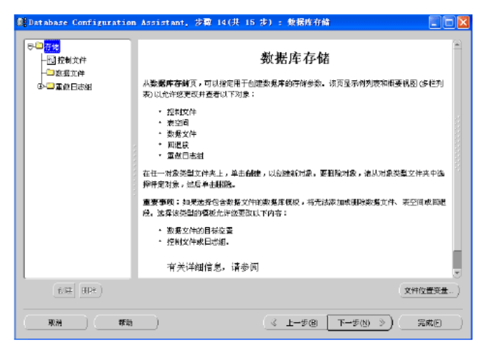


图 5-19　数据库存储参数设置

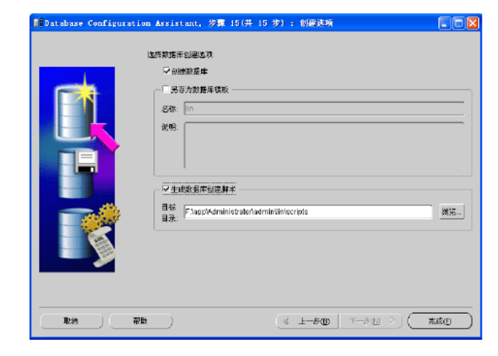


图 5-20　数据库创建选项

19 在图 5-21 中提示建库的详细信息，如初始化参数、字符集、表空间、数据文件、重做日志文件的信息，单击“确定”按钮，则弹出如图 5-22 所示对话框，生成建库的脚本文件，单击“确定”按钮后开始创建数据库，建库过程如图 5-23 所示。



图 5-21　创建数据库的详细信息

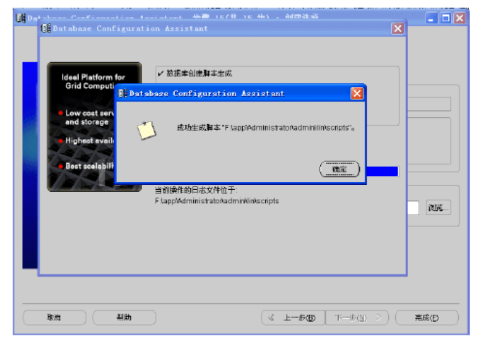


图 5-22　生成创建数据库的脚本

20 当图 5-23 中的创建进度达到 100%时，则弹出如图 5-24 所示的对话框，提供创建数据库的详细信息，如数据库标识、数据库服务器参数文件位置以及账户信息。单击“退出”按钮完成数据库的创建。

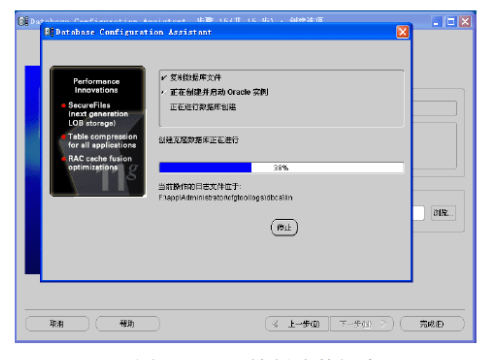


图 5-23　开始创建数据库

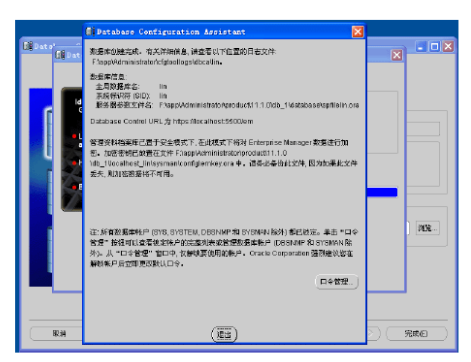


图 5-24　数据库创建完成

5.2.3　理解建库脚本的含义

在使用 DBCA 创建数据库时，依据之前的配置生成了创建数据库的脚本，修改一些参数，如数据库名，然后通过手工执行这些脚本就可以创建数据库。下面通过讲解在 Oracle 10g 中使用 DBCA 创建数据库中生成的脚本来理解 DBCA 创建数据库的过程，首先查看脚本文件的文件目录，如下所示。

```
F:\oracle\product\10.2.0\admin\lin\scripts>dir
 驱动器 F 中的卷是 Oracle10g
 卷的序列号是 000B-1DED

 F:\oracle\product\10.2.0\admin\lin\scripts 的目录

2009-09-15  08:42    <DIR>          .
2009-09-15  08:42    <DIR>          ..
2009-09-15  08:43               338 context.sql
2009-09-15  08:43             1,176 CreateDB.sql
2009-09-15  08:43               717 CreateDBCatalog.sql
2009-09-15  08:43               339 CreateDBFiles.sql
2009-09-15  08:43               192 cwmlite.sql
2009-09-15  08:43               292 emRepository.sql
2009-09-15  08:43             2,488 init.ora
2009-09-15  08:43               187 interMedia.sql
2009-09-15  08:43               408 JServer.sql
2009-09-15  08:43               790 lin.bat
2009-09-15  08:43             1,247 lin.sql
2009-09-15  08:43               191 odm.sql
2009-09-15  08:43               196 ordinst.sql
2009-09-15  08:43             1,214 postDBCreation.sql
2009-09-15  08:43               180 spatial.sql
2009-09-15  08:43               371 xdb_protocol.sql
              16 个文件         10,326 字节
               2 个目录 64,568,418,304 可用字节
```

如果是手工创建数据库，则首先需要运行 lin.bat 文件，下面看一下该文件的内容，如图 5-25 所示。

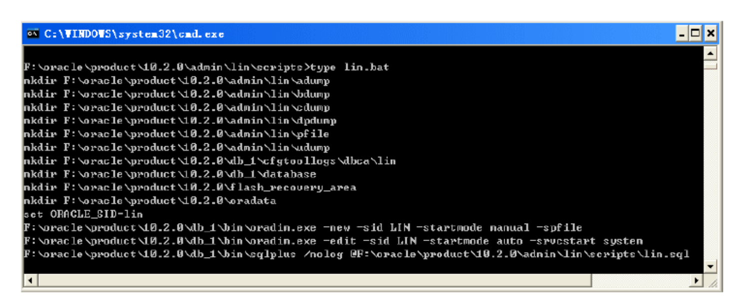


图 5-25　查看脚本文件 lin.bat 的内容

可见，该文件首先是创建一系列目录，然后设置数据库名，通过 oradim.exe 工具配置实例，

然后启动 SQL*Plus 执行 lin.sql 脚本文件创建数据库。

继续查看 lin.sql 的内容，如图 5-26 所示。在该文件的开始部分用于定义用户口令，然后使用 SYS 用户的口令创建口令文件。

```
C:\WINDOWS\system32\cmd.exe

F:\oracle\product\10.2.0\admin\lin\scripts>type lin.sql
set verify off
PROMPT specify a password for sys as parameter 1;
DEFINE sysPassword = &1
PROMPT specify a password for system as parameter 2;
DEFINE systemPassword = &2
PROMPT specify a password for sysman as parameter 3;
DEFINE sysmanPassword = &3
PROMPT specify a password for dbsnmp as parameter 4;
DEFINE dbsnmpPassword = &4
host F:\oracle\product\10.2.0\db_1\bin\orapwd.exe file=F:\oracle\product\10.2.0\db_1\database\PWDlin.ora password=&&sysP
assword force=y
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDB.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBFiles.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBCatalog.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\JServer.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\odm.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\context.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\xdb_protocol.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\ordinst.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\interMedia.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\cwmlite.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\spatial.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\emRepository.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\postDBCreation.sql

F:\oracle\product\10.2.0\admin\lin\scripts>
```

图 5-26　查看脚本文件 lin.sql 的内容

接着将执行一系列的脚本文件，如下所示。

```
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDB.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBFiles.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBCatalog.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\JServer.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\odm.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\context.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\xdb_protocol.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\ordinst.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\interMedia.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\cwmlite.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\spatial.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\emRepository.sql
@F:\oracle\product\10.2.0\admin\lin\scripts\postDBCreation.sql
```

下面主要介绍几个重要的脚本文件。

1. CreateDB.sql

该脚本文件用于创建数据库，内容如下所示。首先使用 SYS 用户登录数据库，该用户具有 SYSDBA 权限。然后启动数据库实例，此时使用参数脚本文件中的初始化参数文件 init.ora，而后创建数据库。

```
F:\oracle\product\10.2.0\admin\lin\scripts>type createdb.sql
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool F:\oracle\product\10.2.0\admin\lin\scripts\CreateDB.log
startup nomount pfile="F:\oracle\product\10.2.0\admin\lin\scripts\init.ora";
CREATE DATABASE "lin"
MAXINSTANCES 8
MAXLOGHISTORY 1
```

```
MAXLOGFILES 16
MAXLOGMEMBERS 3
MAXDATAFILES 100
DATAFILE SIZE 300M AUTOEXTEND ON NEXT  10240K MAXSIZE UNLIMITED
EXTENT MANAGEMENT LOCAL
SYSAUX DATAFILE SIZE 120M AUTOEXTEND ON NEXT  10240K MAXSIZE
 UNLIMITED
SMALLFILE DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE SIZE 20M
AUTOEXTEND ON NEXT  640K MAXSIZE UNLIMITED
SMALLFILE UNDO TABLESPACE "UNDOTBS1" DATAFILE SIZE 200M
AUTOEXTEND ON NEXT  5120K MAXSIZE UNLIMITED
CHARACTER SET ZHS16GBK
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1  SIZE 51200K,
GROUP 2  SIZE 51200K,
GROUP 3  SIZE 51200K
USER SYS IDENTIFIED BY "&&sysPassword" USER SYSTEM IDENTIFIED BY
"&&systemPassword";
set linesize 2048;
column ctl_files NEW_VALUE ctl_files;
select concat('control_files=''', concat(replace(value, ', ', ''','''), ''''))
ctl_files from v$parameter
 where name ='c
ontrol files';
host "echo &ctl_files >>F:\oracle\product\10.2.0\admin\lin\scripts\init.ora";
spool off
```

最后将控制文件信息注册到初始化参数文件 init.ora 中，并且该脚本文件的执行过程都记录在文件 F:\oracle\product\10.2.0\admin\lin\scripts\CreateDB.log 中。

2. CreateDBFiles.sql

该脚本文件的作用是创建数据库数据文件的表空间，并且将该表空间作为用户创建的数据库对象的默认存储空间，其内容如下所示。

```
F:\oracle\product\10.2.0\admin\lin\scripts>type createdbfiles.sql
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBFiles.log
CREATE SMALLFILE TABLESPACE "USERS" LOGGING DATAFILE SIZE 5M
AUTOEXTEND ON NEXT  1280K MAXSIZE UNLIMITED EXTENT MANAGEME
NT LOCAL SEGMENT SPACE MANAGEMENT  AUTO;
ALTER DATABASE DEFAULT TABLESPACE "USERS";
spool off
```

上述输出说明表空间 USERS 为小文件表空间（在 Oracle 10g 中与大对象表空间对应）以及表空间的一些参数设置，如区段管理方式、表空间增长方式以及段管理方式等。

3. CreateDBCatalog.sql

该脚本文件主要用于创建数据库的数据字典、创建 PL/SQL 所需要的软件包和过程，以及用户

的概要文件和相关过程等信息。该文件的内容如下所示。

```
F:\oracle\product\10.2.0\admin\lin\scripts>type CreateDBCatalog.sql
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool F:\oracle\product\10.2.0\admin\lin\scripts\CreateDBCatalog.log
@F:\oracle\product\10.2.0\db_1\rdbms\admin\catalog.sql;
@F:\oracle\product\10.2.0\db_1\rdbms\admin\catblock.sql;
@F:\oracle\product\10.2.0\db_1\rdbms\admin\catproc.sql;
@F:\oracle\product\10.2.0\db_1\rdbms\admin\catoctk.sql;
@F:\oracle\product\10.2.0\db_1\rdbms\admin\owminst.plb;
connect "SYSTEM"/"&&systemPassword"
@F:\oracle\product\10.2.0\db_1\sqlplus\admin\pupbld.sql;
connect "SYSTEM"/"&&systemPassword"
set echo on
spool F:\oracle\product\10.2.0\admin\lin\scripts\sqlPlusHelp.log
@F:\oracle\product\10.2.0\db_1\sqlplus\admin\help\hlpbld.sql helpus.sql;
spool off
spool off
```

4. emRepository.sql

该脚本文件用于创建 EM（企业管理器）的档案库，该文件的内容如下所示：

```
F:\oracle\product\10.2.0\admin\lin\scripts>type emRepository.sql
connect "SYS"/"&&sysPassword" as SYSDBA
set echo off
spool F:\oracle\product\10.2.0\admin\lin\scripts\emRepository.log
@F:\oracle\product\10.2.0\db_1\sysman\admin\emdrep\sql\emreposcre
F:\oracle\product\10.2.0\db_1 SYSMAN &&sysmanPassword
TEMP ON;
WHENEVER SQLERROR CONTINUE;
spool off
```

5. postDBCreation.sql

该脚本文件用于完成数据库创建以后的工作，如创建 SPFILE 参数文件、修改用户口令、配置监听器等。该脚本文件的内容如下所示：

```
F:\oracle\product\10.2.0\admin\lin\scripts>type postDBCreation.sql
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool F:\oracle\product\10.2.0\admin\lin\scripts\postDBCreation.log
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
create spfile='F:\oracle\product\10.2.0\db_1\database\spfilelin.ora' FROM
pfile='F:\oracle\product\10.2.0\admin\lin\scri
pts\init.ora';
shutdown immediate;
connect "SYS"/"&&sysPassword" as SYSDBA
startup ;
alter user SYSMAN identified by "&&sysmanPassword" account unlock;
```

```
alter user DBSNMP identified by "&&dbsnmpPassword" account unlock;
select 'utl_recomp_begin: ' || to_char(sysdate, 'HH:MI:SS') from dual;
execute utl_recomp.recomp_serial();
select 'utl_recomp_end: ' || to_char(sysdate, 'HH:MI:SS') from dual;
host F:\oracle\product\10.2.0\db_1\bin\emca.bat -config dbcontrol db -silent
-DB UNIQUE NAME lin -PORT 1521 -EM HOME F:\
oracle\product\10.2.0\db_1 -LISTENER LISTENER -SERVICE_NAME lin -SYS_PWD
&&sysPassword -SID lin -ORACLE_HOME F:\oracle\p
roduct\10.2.0\db_1 -DBSNMP_PWD &&dbsnmpPassword -HOST localhost
-LISTENER OH F:\oracle\product\10.2.0\db 1 -LOG FILE F:\
oracle\product\10.2.0\admin\lin\scripts\emConfig.log -SYSMAN_PWD
&&sysmanPassword;
spool F:\oracle\product\10.2.0\admin\lin\scripts\postDBCreation.log
```

这里主要是告诉读者脚本就是手动建库的 SQL 指令操作的集合，自动建库就是将手工建库的过程通过各种 SQL 文件自动化了，这样使得读者可以更好地理解 DBCA 自动建库的过程。

5.3　使用安装程序自动创建数据库

除了使用 DBCA 图形化工具创建数据库外，也可以在安装数据库软件时创建数据库，如图 5-27 所示为安装数据库软件的一个界面。选中“创建启动数据库”复选框，然后设置数据库用户的密码，注意此时密码对用户 SYS、SYSTEM、SYSMAN 和 SYSNMP 有效，如果需要其他参数设置，可以选中“高级安装”单选按钮。

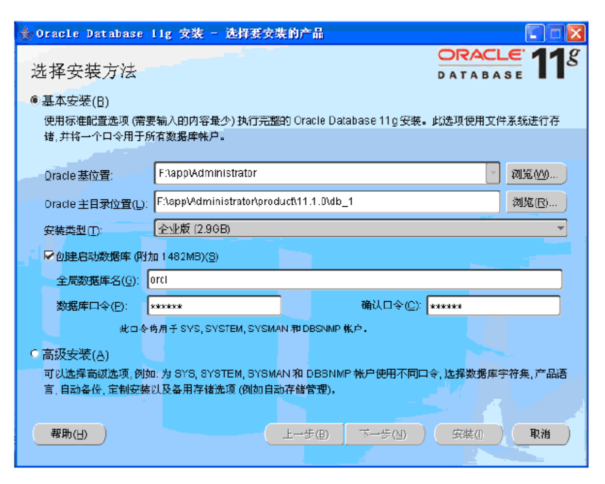


图 5-27　安装数据库软件并创建数据库

然后单击“下一步”按钮，如图 5-28 所示。此时要检查安装数据库软件的一些前提条件是否满足，单击“下一步”按钮，此时会提示一些条件检查失败信息，单击“确定”按钮即可，然后执行一些相关性的分析处理，最后单击图 5-28 中的“下一步”按钮弹出如图 5-29 所示的安装对话框开始安装数据库软件。



图 5-28　检查安装数据库软件的条件



图 5-29　开始安装数据库软件

在数据库软件安装完毕后，则开始创建数据库，此时发现 Oracle 还是使用 DBCA 来完成数据库的创建任务的，如图 5-30 所示。

一旦数据库创建完成，则给出如图 5-31 所示的提示信息。单击“确定”按钮则完成了数据库软件的安装并创建了数据库

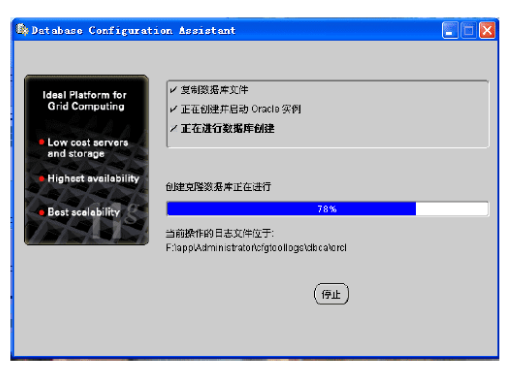


图 5-30　创建数据库

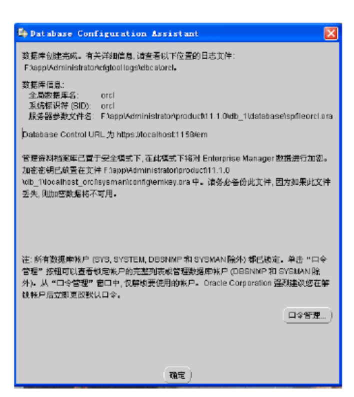


图 5-31　建库完成后的提示信息

在采用这种方式创建数据库时，步骤很简洁，但是所有的控制文件、数据文件和重做日志文件都存放在$ORACLE_HOME\oradata\<oracle_sid>目录下，在建库后需要手工多路复用控制文件和重做日志文件，以提高数据库的可靠性。

5.4　手工创建数据库

手工创建数据库需要详细的前期工作，并且要求对操作系统和应用系统有清晰地了解，手工创建数据库的过程如下。

01 确定唯一的实例名和数据库名。

02 选择数据库字符集。

03 设置操作系统变量。

04 编辑或创建初始化参数文件。

05 启动实例（NOMOUNT）。

06 执行 CREATE DATABASE 指令。

07 运行脚本来创建数据字典并完成之后的数据库创建过程。

下面将以上的步骤做详细的介绍。

1. 确定实例名

首先需要确定数据库名和实例名，一般在设置时将二者的名字设为相同以便于维护，一个数据库可以对应多个实例，如集群 RAC 系统，但是一个实例只能对应一个数据库。

2. 选择数据库字符集

要根据操作系统选择字符集，如果是中文 Windows 系统，则最好选用中文字符集 ZHS16GBK，这样可以减少数据库字符集和操作系统字符集之间的转换。

3. 设置操作系统变量

在创建数据库前要设置如下所示的操作系统变量。

- ORACLE_BASE：位于 Oracle 软件的顶层目录。在笔者的计算机中，这个顶层目录为 F:\oracle。
- ORACLE_HOME：设置 Oracle 软件的安装目录。在笔者的计算机中，该安装目录为 F:\oracle\product\10.2.0。
- ORA_NLS33：当创建的不是 US7ASCII 字符集的数据库时，使用该操作系统参数，如 $ORACLE_HOME/ocommon/nls/admin/data。
- PATH：设置操作系统搜索可执行文件的目录，如执行 SQL*Plus 或 DBCA 等工具，在 Oracle 11g 中该目录是$ORACLE_HOME/bin，并且需要将该目录添加到操作系统的 PATH 变量中。
- LD_LIBRARY_PATH：说明操作系统和 Oracle 数据库文件的目录，笔者的计算机中为 $ORACLE_HOME/lib。

4. 创建初始化参数文件

在安装数据库软件时生成了一个初始化参数文件 init.ora，此时创建一个新库需要的初始化参数文件为 newinit.ora，将 init.ora 文件中的内容全部拷贝到 newinit.ora 文件中，然后修改 newinit.ora 文件中的参数选项，如修改数据库名 db_name 或 SGA 的大小 sga_target 参数等。在参数文件中至少有一个数据库名参数，其他参数可以没有。

此时如果需要使用 SPFILE 参数文件启动数据库，可以使用如下命令初始化 SPFILE 参数文件，即 CREATE SPFILE FORM PFILE。

5. 启动数据库到 NOMOUNT 状态

此时启动了实例，读取参数文件，在当前数据库状态下执行数据库创建，如下所示。

```
SQL> connect system/oracle@orcl as sysdba
```

6. 使用 CREATE DATABASE 指令创建数据库

我们首先给出文献中提供的手工创建数据库的语法格式，然后给出一个实例进行详细介绍。

创建数据库的语法格式如下所示：

```
CREATE DATABASE [database]
  [CONTROLFILE REUSE]
  LOGFILE [GROUP integer]] filename
  [MAXLOGFILES integer]
[MAXLOGMEMBERS integer]
 [MAXLOGHISTORY integer]
[MAXDATAFILES integer]
[MAXDINSTANCES integer]
[ARCHIVELOG|NOARCHIVELOG]
[CHARACTER SET charset]
[NATIONAL CHARACTER SET charset]
[DATAFILE filename [autoextend_clause]]
```

下面给出一个实例说明如何使用 CREATE DATABASE 来创建数据库。

```
SQL> CREATE DATABASE LIN
  2  LOGFILE
  3  GROUP 1 ('F:/logfile/redo01_lin.log') size 20M
  4  GROUP 1 ('F:/logfile/redo02_lin.log') size 20M
  5  GROUP 1 ('F:/logfile/redo03 lin.log') size 20M
  6  MAXLOGFILES 5
  7  MAXLOGMEMBERS 5
  8  MAXLOGHISTORY 8
  9  MAXDATAFILES 256
 10  MAXINSTANCES 1
 11  ARCHIVELOG
 12  FORCE LOGGING
 13  DATAFILE 'F:/LINDATA/SYSTEM01_LIN.DBF' SIZE 500m
 14  UNDO TABLESPACE UNDOTBS
 15  DATAFILE 'F:/UNDODATA/UNDO01_LIN.DBF' SIZE 100m
 16  DEFAULT TEMPORARY TABLESPACE temp
 17  TEMPFILE 'F:/TEMPFILE/temp01_lin.dbf' SIZE 100M
 18  EXTENT MANAGEMENT LOCAL UNIFORM SIZE 1M
 19  character set ZHS16GBK;
```

对以上代码的说明如下。

- 第 1 行：创建名为 LIN 的数据库。
- 第 2~5 行：创建 3 个重做日志组，每个日志组包括一个重做日志成员，每个成员大小为 20M。
- 第 6 行：该数据库中最多有 5 个重做日志组。
- 第 7 行：每个重做日志组中最多有 5 个重做日志成员。
- 第 8 行：在集群环境中自动介质恢复时需要的最多归档日志文件数量。
- 第 9 行：控制文件中保留的数据文件的记录个数。

- 第 10 行：可以同时打开的数据库个数为 1。
- 第 11 行：新建的数据处于归档模式。
- 第 12 行：强制将除临时表空间和临时段中的变化数据外的所有变化记录到重做日志文件中。
- 第 13 行：新数据库所使用的数据文件为 F:/LINDATA/SYSTEM01_LIN.DBF，大小为 500M，注意此时该文件是系统 SYSTEM 表空间基于的数据文件，默认是用户创建的表或其他数据库对象将保存在系统表空间中，所以在创建数据库后要创建一个 USERS 表空间用来存储用户数据。
- 第 14~15 行：设置 UNDO 表空间（还原表空间），该表空间基于的数据文件为 F:/UNDODATA/UNDO01_LIN.DBF，文件大小为 100M。
- 第 16~17 行：设置 TEMP 表空间（默认临时表空间），该表空间基于的数据文件为 F:/TEMPFILE/temp01_lin.dbf，文件大小为 100M。
- 第 18 行：临时表空间的存储参数。该表空间为本地管理的表空间，EXTENT 大小为 1M。
- 第 19 行：设置数据库字符集为中文字符集 ZHS16GBK。



在上述手工创建数据库的过程中没有创建用户，此时使用数据库默认的用户名和密码，其中 SYS 用户的密码为 change_on_install，而 SYSTEM 用户的密码为 manager。

此时，数据库中包含了数据文件、控制文件、重做日志文件、一个数据表空间、一个还原表空间以及一个临时表空间。但是还需要使用脚本文件来创建数据字典。

7. 创建数据字典视图

此时使用 catalog.sql 脚本文件，目录$ORACLE_HOME\RDBMS\ADMIN 为该脚本文件的存储目录。如果读者查看该文件的内容会发现一系列创建视图的 SQL 语句，Oracle 利用脚本文件使得 SQL 指令成批地执行，在执行该脚本文件中创建数据字典时要确保数据库处于打开状态，如下所示。

```
SQL> CONNECT SYSTEM/ORACLE@ORCL AS SYSDBA
SQL>@F:\oracle\product\10.2.0\db_1\RDBMS\ADMIN\CATALOG.SQL;
```

8. 创建 PL/SQL 的软件包和过程

此时使用 catproc.sql 脚本文件。目录$ORACLE_HOME\RDBMS\ADMIN 为该脚本文件的存储目录。在数据库打开的状态下执行该脚本，如下所示：

```
SQL> CONNECT SYSTEM/ORACLE@ORCL AS SYSDBA
SQL>@F:\oracle\product\10.2.0\db_1\RDBMS\ADMIN\CATPROC.SQL;
```

9. 创建用户的概要文件以及相关过程

此时使用 pupbld.sql 脚本文件。目录$ORACLE_HOME\SQLPLUS\ADMIN 为该脚本文件的存储目录。该文件必须在 DBA 用户下执行，如下所示：

```
SQL> CONNECT SYSTEM/ORACLE@ORCL AS SYSDBA
SQL>@F:\oracle\product\10.2.0\db_1\RDBMS\SQLPLUS\CATPROC.SQL;
```



此时已成功手工创建了数据库，但是在创建数据库的过程中会出现一些问题，如创建的文件可能已经存在，此时必须手工删除这些文件再重新创建数据库，如果建库过程中出现了问题，且需要使用 CREATE DATABASE 创建数据库，则要先删除操作系统上已经创建的文件。如遇到文件权限或磁盘空间不足的情况，也需要重新执行相关指令，所以在手工创建数据库前要做详细地“调查研究”，搞清楚需求和机器的硬软件资源。

5.5 本章小结

本章主要讲解了如何创建 Oracle 数据库，在创建数据库之前用户需要了解数据库的业务类型、业务需求和安装数据库的计算机的软硬件配置信息，在条件允许的情况下开始使用图像化工具或手工创建数据库，一般如果是为了学习，可以使用手工方式，因为这对于理解数据库创建的内部机制很有帮助，如果是创建生产数据库，则最好使用图形化工具，如 DBCA 工具，这样很多参数的设置、目录的创建等都能自动完成，从而提高了创建数据库的可靠性和效率。

第 6 章

管理和维护表

管理和维护表在数据库设计、数据库维护中都有很重要的应用，表是数据库中最基本、最常用的存储数据的存储结构。本章将讲解存储在表中的数据类型、如何创建和维护表、表的参数维护和列的维护，最后还将讲解分区表的概念。

6.1 表的概述

表是 Oracle 数据库中最基本的数据存储结构，表是一种逻辑结构，数据在表中以行和列的形式存储。在创建表时，用户需要设定表名、表内各列的列名和各列的数据类型及数据宽度。数据类型包括 VARCHAR2、DATE、NUMBER 或 BLOG 等。在表中同一行的所有列信息叫做记录。下面主要介绍 Oracle 的数据类型。

在 Oracle 的相关文档中，Oracle 定义了三种数据类型，即标量数据类型、集合数据类型和关系数据类型，图 6-1 是 Oracle 中数据类型的关系图。

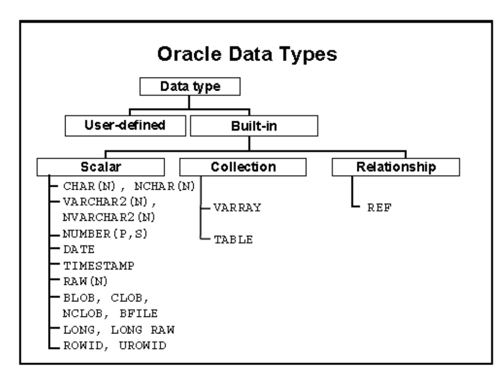


图 6-1 Oracle 的数据类型

下面依次介绍标量数据类型（Scalar）、集合数据类型（Collection）、关系数据类型（Relationships）的相关知识，并介绍一种特殊的数据类型——ROWID。

1. 标量数据类型

（1）VARCHAR2(size)和 NVARCHAR2(size)

首先，该数据类型存储变长的字符数据，在使用该数据类型定义数据时，该数据的存储区大小是不固定的，依据存储数据的长度进行动态分配存储区。参数 size 是该变量存储的最大字符数，该值最大为 4000。size 的最小或默认值都是 1。一般在定义该数据类型时，都要指定该长度值，即指定 size 值。NVARCHAR2(size)的不同之处在于它支持全球化数据类型、支持定长和变长字符集。

（2）CHAR(size) NCHAR(size)

该数据类型一旦定义，则存储该变量的存储区的大小就固定下来了。显然在存储区分配上它没有 VARCHAR2(size)和 NVARCHAR2(size)数据类型具有动态性，但是在实际中，如果可以预测到一个变量存储的字符数量，且数量不是很大，则最好还是使用定长字符型数据来定义该变量，这样可以提高存储的效率。因为使用变长字符型数据要不断地计算存储的数据长度，再分配存储数据块，会消耗计算资源。定长字符型数据的最小值和默认值都为 1 个字符，而最大值为 2000。NCHAR(size)的不同之处在于它支持全球化数据类型，支持定长和变长字符集，此时定长字符型数据的最小和默认值都为 1 个字节。

（3）DATE

Oracle 服务器使用 7 个定长的存储区存储日期型数据，它可以是月、年、日、时、分和秒。日期型数据的取值范围从公元前 4712 年 1 月 1 日到公元 9999 年 12 月 31 日。

（4）NUMBER(p,s)

参数 p 指十进制数中的长度，s 为该十进制数小数点后的位数，如 NUMBER(10,2)表示该数字型数据中的长度为 10 位，而小数后为 2 位。其中参数 p 的最大值为 38，最小值为 1，而参数 s 的最大值为 124，最小值为-84。

（5）CLOB 或 LONG

用于存储大数据对象，该对象为定长的字符型数据，如学术论文或个人简历等。对于 CLOB 数据类型的列的操作不能直接使用 Oracle 数据库指令，需要一个 DBMS_LOB 的 PL/SQL 软件包来维护该数据类型的列。

（6）BLOB 或 LONG RAW

存储无结构的大对象，如照片、PPT、二进制图像等。它和 CLOB 数据类型一样，BLOB 数据类型的列的操作只能通过 PL/SQL 软件包 DBMS_LOB 来实现。

（7）BFILE

在操作系统文件中存储无结构的二进制对象，显然它是 Oracle 的外部数据类型，Oracle 无法直接维护这些数据类型，必须由操作系统来维护。

（8）RAW

该数据类型使得数据库可以直接存储二进制数据，在计算机之间传输该类型数据时，数据库

不对数据做任何转换，所以该数据类型的存储和传输效率较高，RAW 数据类型的最大长度为 2000 个字节。



LONG 和 LONG RAW 数据类型主要用在 Oracle 8 以前的数据库系统。LONG 数据类型完全可以用 Oracle 9i 或 Oracle 10g 的 CLOG 或 BLOB 数据类型替换。

2. 集合数据类型

Oracle 集合数据类型包括嵌套表和 VARRY 数据类型。在嵌套表的列值中又包含表，嵌套表中的元素数量没有限制，当然不能超过表所在的表空间的大小，而 VARRY 集合中的元素是有数量限制的。

3. 关系数据类型

关系类型 REF 指向一个对象，在 Oracle 数据库中一个典型的 REF 类型的对象就是游标 cursor。

4. ROWID

ROWID 也是一种数据类型，但是这种数据类型是 Oracle 服务器使用并管理的。首先解释 ROWID 的特性，通过特性可以理解 ROWID 的作用。

- ROWID 是数据库中每一行的唯一标识符。
- ROWID 作为列值是隐式存储的。
- ROWID 不直接给出行的物理地址，但是可以用 ROWID 来定位行。
- ROWID 提供了最快速地访问表中行的方法。

虽然 ROWID 是非显式存储的，但是对于每一个表，都可以查询该表中每一行的 ROWID。下面通过实例查看 ROWID，并解释 ROWID 的组成和各组成部分的含义，如实例 6-1 所示。

【实例 6-1】登录数据库并查看表 DEPT 的行 ID（ROWID）。

```
C:\>sqlplus /nolog

SQL*Plus: Release 9.0.1.0.1 - Production on 星期四 6月 25 00:19:17 2009

(c) Copyright 2001 Oracle Corporation.  All rights reserved.

SQL> conn scott/tiger;
已连接。
SQL> select deptno,dname,loc,rowid
  2  from dept;

DEPTNO DNAME          LOC           ROWID
------------- ------------------------ --------------------- ------------
      10 ACCOUNTING   NEW YORK     AAAH4EAAIAAAAByAAA
      20 RESEARCH     DALLAS       AAAH4EAAIAAAAByAAB
      30 SALES        CHICAGO      AAAH4EAAIAAAAByAAC
      40 OPERATIONS   BOSTON       AAAH4EAAIAAAAByAAD
```

```
        50 Marcketing                        AAAH4EAAIAAAAByAAE
```

我们选择 DEPTNO 为 20 的行，即输出第二行的 ROWID 来分析，前 6 位 AAAH4E 为数据对象号，在数据库中每个对象是唯一的。接着三位 AAI 为相对文件号，它和表空间中的一个数据文件对应。接着 6 位 AAAABy 为块号，块号为相对文件中存储该行的块的位置，最后 3 位 AAB 为行号，行号标识块头中行目录的位置，而使用该行目录的位置可以找到行的起始地址。



本节只要求读者对于 ROWID 有个了解，知道用 ROWID 可以唯一标识一行、Oracle 使用 ROWID 可以快速找到一行数据即可。

6.2 创建表

本节从实用的角度介绍如何创建一个表以及如何使创建的表易于管理。我们先介绍 Oracle 创建表的规则，并通过实例说明如何创建一个表。

Oracle 数据库推荐了如下一些与表相关的标准，读者在实际中最好使用这些标准，对于维护数据库表和顺利建表都很有好处。

- 命名尽量简单，表名要具有一定的意义，即表名要清楚描述表中存储的数据内容，如一个临时员工表的表名为 temp_employees。
- 每个表对应一个表空间，这样易于管理和维护，对一个表空间的维护不影响其他的表，并且该表空间是本地管理的。
- 使用标准 EXTENT 尺寸减少表空间碎片。
- Oracle 数据库允许表名的最大长度为 30 个字符。

6.2.1 创建普通表

创建数据库的目的是存储数据，而这些数据就存储在表中，表是数据库中最基本的数据存储机制。下面我们使用 Create Table 指令来创建表。

要创建表用户必须拥有创建表的属性，此时，我们使用 dba 用户登录数据库服务器，如实例 6-2 所示。

【实例 6-2】使用 dba 用户登录数据库服务器。

```
SQL> conn /as sysdba
已连接。
```

接下来创建一个临时员工表，该员工表属于 SCOTT 用户，并且存储在 USERS 表空间中，如实例 6-3 所示。

【实例 6-3】创建一个临时员工表 temp_employees。

```
SQL> create TABLE scott.temp_employees
```

```
  2        (employee_id        number(6),
  3         employee_name      varchar(30),
  4         employee_sex       char,
  5         department        varchar(30))
  6         TABLEspace users;

表已创建。
```

一旦使用 DDL 语句创建了表对象，对象的信息，如表名、表存储的表空间等将记录在数据字典中，数据字典将在第 7 章讲解，这里只需要读者知道这个概念，实例 6-3 中创建表的信息将记录在数据字典 dba_tables 中。下面使用实例 6-4 验证是否成功创建该表。

【实例 6-4】验证实例 6-3 中是否成功创建表 temp_employees。

```
SQL> select owner,table name,tablespace name
  2  from dba_tables
  3  where owner = 'SCOTT';

OWNER                    TABLE NAME          TABLESPACE NAME
------------------------------ ------------------------------
SCOTT                       BONUS                   USERS
SCOTT                       DEPT                    USERS
SCOTT                       DEPT_TEMP               SYSTEM
SCOTT                       EMP                     USERS
SCOTT                       EMP_TEMP
SCOTT                       ORD                     SYSTEM
SCOTT                       PRODUCT                 SYSTEM
SCOTT                       SALGRADE                USERS
SCOTT                       SUPPLIER                SYSTEM
SCOTT                       TEMP_EMPLOYEES          USERS

已选择 10 行。
```

输出结果的最后一行说明，已经成功创建了表 temp_employees，该表所属的用户为 SCOTT，而存储该表的表空间为 USERS。


说　明

如果在创建表时不指定用户名字，直接写表名，则默认是当前用户创建的表，如果不指定表空间名，则 Oracle 将使用默认表空间创建该表。

在创建表的原则中，Oracle 推荐了一个表最好放在一个表空间而且该表空间是本地管理的（减少维护数据字典的负担），所以如果已经创建了一个本地管理的表空间可以使用更多的参数在本地管理的表空间中创建表，在 Oracle 10g 中创建的表空间，本地管理是默认方式，如实例 6-5 所示，先创建一个本地管理的表空间 lin，然后再在该表空间中创建一个表。

【实例 6-5】创建一个本地管理的表空间 lin。

```
SQL> create TABLEspace lin
  2  datafile 'd:\temp\lin.dbf'
```

```
  3 size 30M
  4 extent management local
  5 uniform size 1M;

表空间已创建。
SQL> create TABLE scott.employees
  2        (ecode    number(6),
  3         ename    varchar2(25),
  4         eaddress varchar2(30),
  5         ephone   varchar2(15))
  6 storage (initial 100k next 100k pctincrease 0 minextents 1
  7 maxextents 8)
  8* TABLEspace lin

表已创建。
```

这里需要解释 storage 中的参数含义，initial 是指对于该表而言，当表的数据量增加时，需要自动分配磁盘空间时第一次分配 100k，第二次也是 100k，所分配的最大磁盘为 8 个 EXTENTS。最小为 1 个 EXTENTS，而 PCTINCREASE 是一个权值参数，是指当第三次为该表增加磁盘空间时，需要按规则计算：$NEXT*(1+PCTINCRASE/100)^{(n-2)}$，其中 n>=3，即如果第三次需要增加磁盘空间时，分配 $100*(1+0/100)^{(3-2)}$=100k，第四次需要增加磁盘空间时，分配 $100*(1+0/100)^{(4-2)}$=100k，可以看出如果选择 PCTINCRASE 为 0，则每次分配的磁盘空间和 NEXT 参数值相同。

下面使用实例 6-6 来验证是否成功建立表 employees。

【实例 6-6】验证是否成功建立表 employees。

```
SQL> select TABLE_name,TABLEspace_name ,initial_extent,next_extent
  2 from dba TABLEs
  3 where owner = 'SCOTT'
  4 and TABLE_name = 'EMPLOYEES';

TABLE_NAME    TABLESPACE_NAME    INITIAL_EXTENT    NEXT_EXTENT
------------------------- ----------------------------------
EMPLOYEES      LIN                       102400         1048576
```

上述输出说明，表 employees 已经成功创建，并且在表空间 LIN 中，该表的参数 INITIAL_EXTENT 为 100k(102400/100 = 100k)，而且 NEXT_EXTENT 也为 100k。

6.2.2 创建临时表

临时表是非常特殊的表，该表只对当前用户的当前会话有效。创建临时表的目的就是使某些操作效率更高。临时表中的数据只对当前会话的用户有效，是当前会话的私有数据，当前会话只操作自己的数据，没有数据锁的争用，这极大提高了临时表操作的效率。下面依次从创建临时表和临时表的可见性方面进行详细介绍。

1.临时表的创建过程

下面通过使用 CREATE GLOBAL TEMPORARY 指令来创建临时表，如实例 6-7 所示，该临

时表为 SCOTT 用户的 EMP 表中所有 JOB 为 MANAGER 的员工信息。

【实例 6-7】使用 CREATE GLOBAL TEMPORARY 指令来创建临时表。

```
SQL>  create global temporary TABLE
  2  scott.emp_temporary
  3  on commit preserve rows
  4  as
  5  select *
  6  from scott.emp
  7  where job = 'MANAGER';

表已创建。
```



该临时表默认存储在系统的临时段中，如果临时表空间为空，也无法创建成功，会有如下错误提示。

```
SQL> create global temporary TABLE
  2  scott.emp temporary
  3  on commit preserve rows
  4  as
  5  select *
  6  from scott.emp
  7  where job = 'MANAGER';
from scott.emp
         *
ERROR 位于第 6 行:
ORA-25153: 临时表空间为空
```

遇到这样的问题，只要新建立一个临时表空间，然后改变系统的临时表空间为新建立的表空间即可。

读者可以使用实例 6-8 来验证是否成功创建该临时表。

【实例 6-8】验证是否成功创建该临时表。

```
SQL> select owner,TABLE_name,TABLEspace_name
  2  from dba_TABLEs
  3  where TABLE_name = 'EMP_TEMPORARY';

OWNER                TABLE_NAME           TABLESPACE_NAME
-------------------- -------------------- ----------------------------
SCOTT                EMP_TEMPORARY
```

输出结果中 TABLESPACE_NAME 列为空，说明临时表并不是存放在默认表空间，也不存放在临时表空间中，而是存储在临时段中，临时段是一个磁盘区，当用户使用 SQL 语句执行查询时，如果需要对返回的数据进行排序时，Oracle 首先需要在内存中完成排序工作，如果内存容量不够，就需要把计算的中间结果放在临时段中。

【实例 6-9】查询表 EMP_TEMPORARY 是否为临时表。

```
SQL> select table_name,tablespace_name,temporary
  2  from dba_tables
  3  where owner = 'SCOTT'
  4  and  table_name = 'EMP_TEMPORARY';

TABLE_NAME          TABLESPACE_NAME             T
------------------------------------ ---------
EMP_TEMPORARY                                    Y
```

实例 6-9 的输出说明，表 EMP_TEMPORARY 为临时表，而且该表没有存放在用户 SCOTT 的默认表空间中，而是存储在临时段中。

2. 临时表的可见性

临时表在当前用户的当前会话下可用。如果用户使用其他用户登录，如使用 dba 用户，或者重新启动了数据库，则无法使用该临时表。实例 6-10 说明了当使用相同的用户名，如 SYS 用户重新登录数据库时，查询临时表 emp_temporary 的输出结果。

【实例 6-10】查询临时表 emp_temporary 的输出结果。

```
SQL> conn /as sysdba
已连接。
SQL> desc scott.emp_temporary;
 名称                                            是否为空? 类型
 ----------------------------------------------------- --------
 EMPNO                                                  NUMBER(4)
 ENAME                                                  VARCHAR2(10)
 JOB                                                    VARCHAR2(9)
 MGR                                                    NUMBER(4)
 HIREDATE                                               DATE
 SAL                                                    NUMBER(7,2)
 COMM                                                   NUMBER(7,2)
 DEPTNO                                                 NUMBER(2)

SQL> select *
  2  from emp_temporary;
from emp temporary
    *
ERROR 位于第 2 行:
ORA-00942: 表或视图不存在
```

上例说明，使用 DBA（用户名为 SYS）用户重新登录数据库，此时可以查看到临时表 emp_temporary 的数据字典定义，但是不能成功查询该表中的数据。也就是说临时表只对当前用户的当前会话有效。一旦用户退出当前会话，则临时表就失去了作用。

如果不再使用临时表，则最好删除掉，毕竟它占用存储空间，而且一旦用户改变或重新登录，都无法重新使用该表，如实例 6-11 所示为删除临时表。

【实例 6-11】删除临时表。

```
SQL> DROP TABLE scott.emp_temporary;

表已丢弃。
```

6.3　维护参数

在创建表时，我们使用了表空间和 storage 参数，如 initial、next、pctincrease 等，而在数据库的维护过程中，这些参数是不允许变化的，但是 Oracle 为了管理和控制数据块而引入了 5 个参数。

1. INTRANS

该参数用于控制对数据块的并行操作的参数。在解释该参数前，先介绍事务槽的概念，事务槽在数据块头中，存储了有关事务的控制信息，而每行数据有一个锁位，该锁位号和事务槽号相同，数据库服务器就是通过每行的锁位找到数据块头中的事务槽，利用存储在该事务槽中的事务信息完成对该行数据的操作。每个事务只使用一个事务槽。锁位和事务槽的关系以及数据块的组成如图 6-2 所示。

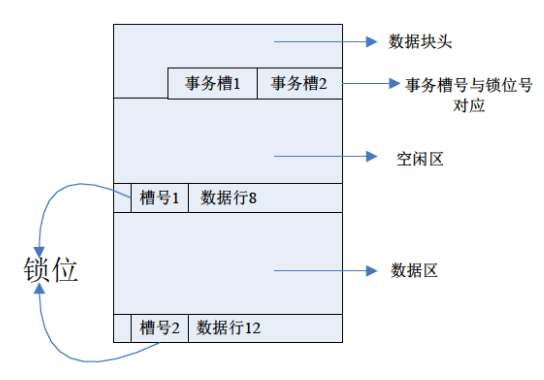


图 6-2　数据块结构及事务槽与锁位关系图

INITRANS 定义了创建数据块时事务槽的初始值，该参数的默认值为 1，如该参数为 2，说明数据库服务器在一个数据块中可以最多有 2 个并行的事务，每个事务独立、并行地通过自己的事务槽，并且实现对该行数据的事务操作。

2. MAXTRANS

该参数定义了创建数据块时事务槽的最大值，该参数的默认值为 255。

下面使用实例 6-12 来查看 SCOTT 用户的表 EMPLOYEES 的表参数 INITRANS 和 MAXTRANS 参数。

【实例 6-12】查看 SCOTT 用户的表 EMPLOYEES 的 INITRANS 和 MAXTRANS 参数。

```
SQL> select ini_trans,max_trans
```

```
  2  from dba_TABLEs
  3  where owner = 'SCOTT'
  4  and TABLE_name = 'EMPLOYEES';

 INI_TRANS  MAX_TRANS
-------------------- ------------------
         1          255
```

输出说明，表 EMPLOYEES 的两个参数 INITRANS 和 MAXTRANS 都采用了默认值。

3. PCTFREE

该参数用于设置每个数据块中预留空间的百分比数。当数据块需要额外空间时使用 PCTFREE 参数设置的空间。该参数的默认值为 10%。为了更好地理解 PCTFREE 参数的含义，先介绍数据块的结构，如图 6-3 所示。

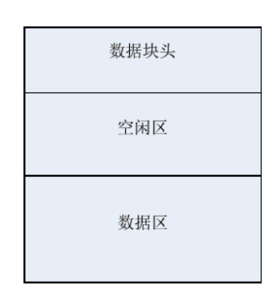


图 6-3　数据块结构图

数据块由上部分组成，即数据块头、空闲区和数据区，其中数据块头从上往下增长，而数据区是从下往上增长，但二者“碰头”则空闲区被占满。

但是一旦空闲区满，而且后续操作如修改数据、增加了数据量，需要占用数据空间，此时该数据块中就无法满足要求，这样就造成占用其他数据块的空间，这种空间的置换会带来磁盘 I/O 的效率低下，所以最好在一个数据块中放置修改的数据。Oracle 设计了 PCTFREE 参数，在每个数据块中设置了一个预留空间，满足在数据操作时对数据块空间的要求。

如果修改的数据行需要额外的空间，就使用 PCTFREE 参数指定的预留大小的空间。

4. FREELISTS

该参数是一个空闲数据块队列的列表，当用户向表中插入数据时，需要数据块作为存储空间，那么该参数中的数据块就作为候选的数据块。

5. PCTUSED

该参数定义了数据块中已经使用的空间的百分比数。如果该数据块中已经使用的空间的百分比低于该参数值时，该数据块才放入段中的空闲数据块列表。该参数的默认值为 40%。

我们先使用实例 6-13 查询 SCOTT 用户的表 EMPLOYEES（该表在 6.2 节已经创建）的参数设置。

【实例 6-13】查询 SCOTT 用户的表 EMPLOYEES 的参数设置。

```
SQL> select TABLE_name,TABLEspace_name,pct_free,pct_used
  2  from dba_TABLEs
  3  where owner = 'SCOTT'
  4* and TABLE_name = 'EMPLOYEES'

TABLE_NAME           TABLESPACE    PCT_FREE    PCT_USED
---------------------------------------- -------------------- --
EMPLOYEES            LIN                10          40
```

上述输出结果显示该表的参数 PCT_FREE 为 10，PCT_USED 为 40。下面使用 ALTER TABLE 指令动态修改表的参数，如实例 6-14 所示。

【实例 6-14】动态修改表的参数 pctused 和 pctfree。

```
SQL> ALTER TABLE scott.employees
  2  pctused 50
  3  pctfree 30;

表已更改。
```

为了验证修改结果，可以再次使用实例 6-15 来查看结果。

【实例 6-15】查看实例 6-14 对表参数 pctused 和 pctfree 的修改结果。

```
SQL> select TABLE_name,TABLEspace_name,pct_free,pct_used
  2  from dba TABLEs
  3  where owner = 'SCOTT'
  4  and TABLE_name = 'EMPLOYEES';

TABLE_NAME            TABLESPACE   PCT_FREE   PCT_USED
-------------------- ---------- ---------- ----------
EMPLOYEES             LIN               30         50
```


说 明

由于 Oracle 中参数之间的相互关联性，修改一个参数很可能会影响其他参数的作用，所以，读者在刚开始学习 Oracle 时不要轻易修改系统的各种默认参数。

6.4　维护列

在实际的项目开发过程中，需要对建好的表进行修改，如增加或删除列、修改列名等，这样的操作如果通过重新建表显然是不现实的，本节将讲解如何在已经建好的表中完成列维护。我们在 6.2 节中已经建立了一个表 employees。因为该表没有数据，先向该表中插入数据，如实例 6-16 所示。

【实例 6-16】向表 employees 中插入数据。

```
SQL> insert into scott.employees
  2  values(1,'Tom','address1','80854340');

已创建 1 行。
```

使用实例 6-17 查询该表中的所有数据。

【实例 6-17】查询表 employees 中的所有数据。

```
SQL> select *
  2  from scott.employees;
```

```
ECODE ENAME                                  EADDRESS                            EPHONE
---------- -------------------------------------------- ------
     1 Tom                                    address1                             80854340
```

该表有 4 列，分别为 ECODE、ENAME、EADDRESS 和 EPHONE，该表中只有一行记录，员工名字是 Tom。

1. 插入列

下面我们演示如何插入一列，显然在表 employees 中没有员工性别是不合适的。我们在表中增加一列 SEX，如实例 6-18 所示。

【实例 6-18】向表 employees 中增加一列 SEX。

```
SQL>ALTER TABLE scott.employees
  2   add (
  3       sex    char
  4       );
表已更改。
```

为了验证是否增加了一列，可使用实例 6-19 继续查询该表中的所有数据。

【实例 6-19】验证实例 6-18 是否向表 employees 中增加了一列。

```
SQL> col ename for a10
SQL> col eaddress for a20
SQL> col sex for a10
SQL> select *
  2* from scott.employees

ECODE ENAME      EADDRESS              EPHONE          SEX
---------- -------------------- ---------------------------------- ------
    1 Tom      address1             80854340
```

从输出结果可以看出，我们已经成功添加了一列，该列名为 SEX，但是值为空。

下面我们再增加一列，并且对该列进行修改。列名为 DEGREE，如实例 6-20 所示。

【实例 6-20】向表 employees 中增加一列。

```
SQL> ALTER TABLE scott.employees
  2  add (
  3       degree varchar2(10)
  4       );

表已更改。
```

此时，我们更新表中的数据使得员工 Tom 的 SEX 列和 DEGREE 列都有数据，如实例 6-21 所示。

【实例 6-21】更新员工 Tom 的 SEX 列和 DEGREE 列的值。

```
SQL> update scott.employees
  2   set sex ='m',degree='Bachelor'
```

```
  3  where ename = 'Tom';

已更新 1 行。
```

查询该结果，如实例 6-22 所示。

【实例 6-22】查询实例 6-21 的修改结果。

```
SQL> col sex for a3
SQL> col eaddress for a10
SQL> select *
  2* from scott.employees

ECODE ENAME      EADDRESS   EPHONE          SEX DEGREE
---------- -------------------- ----------------- -----
    1 Tom          address1      80854340          m   Bachelor
```

输出显示已成功在新添加的列中增加了数据。但是如果需要修改一个列的约束，如不允许该列为空（NULL），则需要修改列。

2. 修改列

把列 DEGREE 设置为不允许为空（null），如实例 6-23 所示。

【实例 6-23】将表 employees 中的列 DEGREE 设置为不允许为空。

```
SQL> ALTER TABLE scott.employees
  2  modify(
  3  degree varchar2(10) not null
  4        )
  5  ;

表已更改。
SQL> desc scott.employees;
 名称                                    是否为空? 类型
 ----------------------------------------- -------- ---------------
 ECODE                                              NUMBER(6)
 ENAME                                              VARCHAR2(25)
 EADDRESS                                           VARCHAR2(30)
 EPHONE                                             VARCHAR2(15)
 SEX                                                CHAR(1)
 DEGREE                                    NOT NULL VARCHAR2(10)
```

我们成功修改了列 DEGREE 的约束，不允许该列为空，如果用户再次插入数据时，该列为空则无法成功插入。

3. 删除列

用户既然可以修改表中的列、添加列，自然也可以删除不需要的列。删除列的语法格式为：

```
ALTER TABLE tablename DROP COLUMN columnname CASCADE CONTRAINTS
```

参数 CASCADE CONSTRAINTS 不是必须的，但是如果该列是一个表的外键，也就是说该

表是一个外键引用的父表，且该外键就是此时要删除的列，则需要使用 CASCADE CONSTRAINTS 参数。

该操作对于 Oracle 8i 以上的版本都适用，但是使用该指令时，数据库系统会重新将表写入磁盘，目的是为了还原需要，这样对于一个大表而言就会占用较大的还原空间，一旦删除该列就无法恢复。

【实例 6-24】删除表 employees 中的列 DEGREE。

```
SQL> ALTER TABLE scott.employees DROP COLUMN degree;

表已更改。
SQL> desc scott.employees;
 名称                                    是否为空? 类型
 ----------------------------------------- -------- -------------
 ECODE                                              NUMBER(6)
 ENAME                                              VARCHAR2(25)
 EADDRESS                                           VARCHAR2(30)
 EPHONE                                             VARCHAR2(15)
 SEX                                                CHAR(1)
```

上述输出说明，已成功删除表 employees 中的列 DEGREE。

在大表中删除一行非常耗费时间，此时可以在 ALTER TABLE 语句中使用 SET UNUSED 子句，这样就将表中某列置成无用的列。其语法如下：

```
ALTER TABLE <username.>tablename
SET UNUSED COLUMN columnname CASCADE CONSTRAINTS;
```

其实，此时并没有删除该表中该列的数据，而是使得用户查询时看不到该列内容，该列被数据库认为是删除的列。并且一旦该列设置为 UNUSED 则无法恢复。但数据库空闲时，可以使用如下命令再删除置为无用的列。

```
ALTER TABLE <username.>tablename
DROP UNUSED COLUMNS ;
```

下面演示如何将一列设置成无用的列，如实例 6-25 所示。

【实例 6-25】将表 employees 中的 EPHONE 列设置为无用的列。

```
SQL> ALTER TABLE scott.employees
  2* SET UNUSED  COLUMN ephone

表已更改。
SQL> select *
  2  from scott.employees;

ECODE ENAME      EADDRESS    SEX
---------- -------------------- -------------------- ---
     1 Tom            address1      m
```

表 employees 中的列 EPHONE 设置为无用，此时数据库认为该列已经删除，在使用 SQL 语

句查询时会发现没有列 EPHONE 的值。

下面删除设置为 UNUSED 的列，如实例 6-26 所示。

【实例 6-26】删除表 employees 中设置为 UNUSED 的列。

```
SQL> alter table scott.employees
  2  drop unused columns;

表已更改。
```

4. 更改列名字

列名是程序员设计用来表示一个字段的信息，如果在开发过程中由于某种原因需要对列名进行规范，则需要对已经创建好的表的列名进行修改。修改列名的语句很简单，如下所示。

```
ALTER TABLE <username.>employees
RENAME COLUMN old_columnname
TO new_columnname;
```

实例 6-27 给出了修改列名的实例演示，该实例将表 employees 中的列 SAL 改为 SALARY。

【实例 6-27】将表 employees 中的列 SAL 改为 SALARY。

```
SQL> alter table scott.emp
  2  rename column sal
  3  to salary;

表已更改。
```

查询更改结果，如下例所示。

```
SQL> desc scott.emp;
 名称                                       是否为空? 类型
 ----------------------------------------- --------
 EMPNO                                      NOT NULL NUMBER(4)
 ENAME                                               VARCHAR2(10)
 JOB                                                 VARCHAR2(9)
 MGR                                                 NUMBER(4)
 HIREDATE                                            DATE
 SALARY                                              NUMBER(7,2)
 COMM                                                NUMBER(7,2)
 DEPTNO                                              NUMBER(2)
```

从上例的输出中可以看到原表中的 SAL 列名已经改为 SALARY 了，说明实例 6-27 修改列名成功。

6.5　删除和截断表

在不需要一个表时，可以删除表，即使用 DROP TABLE 语句实现，此时会彻底删除表中的数据和表的结构，表的结构是指在创建表时定义的列名、列属性和一些约束信息。如果只想删除

表中的数据可以使用 TRUNCATE TABLE 语句截断一个表，此时会保留表的结构，只删除表中的数据。本节将讲解和演示如何删除和截断表。

为了演示如何删除或截断一个表，先通过实例 6-28 创建一个表 EMP_TEMP。

【实例 6-28】创建一个表 EMP_TEMP。

```
SQL> create TABLE scott.emp_temp
  2  as
  3  select *
  4  from scott.emp;

表已创建。
```



读者在使用 DROP 或 TRUNCATE 删除表或截断表时，表中的数据是无法恢复的，如果操作本小节的实例，最好如本书中介绍的一样，先创建一个表。

1. 截断表（TRUNCATE）

不严谨地讲，表由表结构和表中的数据组成，如果不需要表中的数据，可以使用 TRUNCATE 来截断一个表。使用该指令截断表时，只删除表中的数据，但是保留表的结构，也就是对表的定义还是存在的，可以使用 INSERT 指令向表中添加数据。

该指令的语法格式为：

```
TRUNCATE TABLE username.TABLEname
```

为了便于读者直观理解，这里给出一个实例，然后再介绍 TRUNCATE 的其他特性。

【实例 6-29】截断表 emp_temp。

```
SQL> truncate TABLE scott.emp temp;

表已截断。
```

输出说明用户 SCOTT 中的表 emp_temp 已经截断，再使用实例 6-30 验证改变表的结构，也就是验证对表的定义是否还存在。

【实例 6-30】验证表 emp_temp 的定义是否还存在。

```
SQL> desc scott.emp_temp;
 名称                                          是否为空? 类型
 -----------------------------------------------------------------------
 EMPNO                                                  NUMBER(4)
 ENAME                                                  VARCHAR2(10)
 JOB                                                    VARCHAR2(9)
 MGR                                                    NUMBER(4)
 HIREDATE                                               DATE
 SAL                                                    NUMBER(7,2)
 COMM                                                   NUMBER(7,2)
 DEPTNO                                                 NUMBER(2)
```

显然，输出结果说明用户 SCOTT 的表 emp_temp 依然存在。再使用实例 6-31 来验证表中是否还有数据。

【实例 6-31】验证表 emp_temp 中是否还有数据。

```
SQL> select *
  2  from scott.emp_temp;

未选定行
```

显然，用户 SCOTT 的表 emp_temp 中没有数据。下面总结一下 TRUNCATE 指令的特性，这些特性来源于 Oracle 的文档。

- 删除表中的数据但是保留了表结构。
- 数据一旦删除就释放数据占有的磁盘空间，并且数据不可恢复。
- 如果表被外键引用，则无法使用 TRUNCATE 删除表中的数据。
- 所有和表相关的索引也被截断。
- 不会触发删除表的删除触发器。

在读者学习了索引和触发器后可以使用实例来验证上述对于 TRUNCATE 指令特性的说明是否正确，这里不再给出具体示例。

2. 删除表（DROP）

如果不再需要表，包括表中的数据和表的结构，则使用 DROP 指令。使用该指令时删除的是表的结构，表中的数据无法恢复，所以在使用该指令前务必要确认不再需要该表。使用 DROP 删除表的语法格式为：

```
DROP TABLE [username.]TABLEname
CASCADE CONSTRAINTS;
```

当一个表被 DROP 后，该表使用的 EXTENTS 得到释放，如果这些 EXTENTS 是连续分配的，则这些连续区域得到释放并组合成更大的可分配空间。如果该表的外键被另一个表引用，即该表在外键关系中是父表，则需要使用 CASCADE CONSTRAINTS，使得该表脱离与子表的关系，并顺利删除。

【实例 6-32】使用 DROP 指令删除表 emp_temp 的表结构。

```
SQL> DROP TABLE scott.emp_temp;

表已丢弃。
```

下面通过实例 6-33 来验证该表的结构和表中数据是否存在。

【实例 6-33】验证 SCOTT 用户的表 emp_temp 结构是否存在。

```
SQL> desc scott.emp_temp;
ERROR:
ORA-04043: 对象 scott.emp_temp 不存在
```

显然用户 SCOTT 的表 emp_temp 已经删除，因为错误提示为"对象 scott.emp_temp 不存在"，当然也无法查询该表中的数据。

6.6 分区表

对于一个很大的表而言，如果每次搜索时都对全表进行扫描，显然会很耗费时间，也降低了系统的效率。Oracle 允许对一个表进行分区，即把大表分解为更容易管理的分区块，按照不同的分区规则将表分布在不同的磁盘上，这样的表实际上是一种逻辑的概念，即用户操作的是一个表，而实际上 Oracle 会到不同分区去搜索数据，如使用 DDL 语句访问一个单独的分区，而不用访问或搜索整个表，分区表对用户而言是透明的，即用户看不到分区的存在，分区由 Oracle 管理，如一个公司有很多子公司分布在世界不同的地方，而建立了一个具有相同属性的表，这个表又是每个子公司需要的，此时这个表就可以设计为分区表，分区表的逻辑关系如图 6-4 所示。

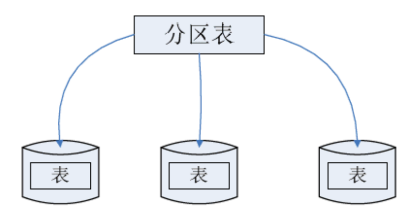


图 6-4　分区表的逻辑模型

在图 6-4 中，分区表由三个分区块组成，每个块组成了分区表的一部分，当操作分区表时不同的用户可以同时操作一个分区表的不同分区块中的数据。

6.6.1 分区表的分类及创建

创建数据库的目的是存储数据，而这些数据就存储在表中，表是数据库中最基本的数据存储机制。下面使用 CREATE TABLE 指令来创建表。

分区表有 4 种不同的分区方法。

1. 范围分区（range partitioning）

根据用户创建分区时指定的键值范围进行表分区，将数据映射到不同的分区。范围分区的键值通常针对日期型数据。

在使用范围分区时，应使用如下的规则：

- 定义分区必须使用 VALUES LESS THAN 子句定义分区的开区间上限。分区键大于或等于此开区间上限的数据存储到下一个分区中。
- 除了第一个分区之外，其他所有分区都有一个隐式的下限，该下限是由上一个分区的 VALUES LESS THAN 子句指定。
- 使用 MAXVALUE 修饰最大分区。MAXVALUE 代表一个无穷大值，用来识别大于所有可能分区键的数据。

为了说明如何创建一个范围分区，下面给出一个具体的实例，如实例 6-34 所示。

【实例 6-34】创建范围分区。

```
CREATE TABLE sales_range(
        Salesman_id number(5),
        Salesman_name varchar2(30),
        Sales amount number(10),
        Sales_date date)
PARTITION BY RANGE(sales_date)(
PARTITION sales_jan2008 VALUES LESS THAN(TO_DATE
('02/01/2008','MM/DD/YYYY')),
PARTITION sales_feb2008 VALUES LESS THAN(TO_DATE
('03/01/2008','MM/DD/YYYY')),
PARTITION sales mar2008 VALUES LESS THAN(TO DATE
('04/01/2008','MM/DD/YYYY')),
PARTITION sales_apr2008 VALUES LESS THAN(TO_DATE
('05/01/2008','MM/DD/YYYY')),
PARTITION sales_may2008 VALUES LESS THAN(TO_DATE
('06/01/2008','MM/DD/YYYY'))
)
```

该分区表的分区键为 sales_date，在创建分区表时，我们使用了 VALUES LESS THAN 子句，通过该子句的参数指定了什么数据放在该分区中。以分区 sales_jan2008 为实例，所有在时间 02/01/2008 之前的数据放在该分区表中，其他分区类似。

2. 列表分区

列表分区显式地将数据行映射到各个分区，这些分区的定义中指定了一个由分区键离散值的列表。显然范围分区与列表分区不同，因为范围分区有一个分区键，而列表分区有多个分区键，是分区键的一个列表，根据这个离散的分区键列表创建表分区。

下面给出一个实例，在该实例中，用户按照区域对销售数据进行分区。把地理位置接近的地区归为一组，如实例 6-35 所示。

【实例 6-35】创建列表分区。

```
CREATE TABLE sales_list(
Salesman_id number(5),
Salesman_name varchar2(30),
Sales_state varchar2(20),
Sales_amount number(10),
Sales date date)
PARTITION BY LIST (sales_state)(
PARTITION sales_west VALUES('California','Hawaii'),
PARTITION sales_east VALUES('New York','Virginia','Florida'),
PARTITION sales centralt VALUES('Texas,'Illinois'),
PARTITION sales_other VALUES(default)
)
```

该分区表有 4 个分区块，键值 sales_state 用来将不同的数据行映射到相应的分区表中，如数据行（25, 'Smith', 'Florida',150, '15-jan-2008'）映射到 sales_east 分区；（23, 'Lee', 'colorado',140,

'21-jan-2008'）映射到 sales_other 分区。在映射过程中，首先确认该行的 sales_state 值，然后再将该值与分区表定义中的分区键值相匹配，找到相应的分区表，将数据插入该分区表。

3. hash 分区

哈希分区是指按照某一个分区键值的哈希函数计算的结果进行分区，在插入某一行数据时，先计算该分区键盘值的哈希函数值，再选择相对应的分区块。

和范围分区与列表分区相比，哈希分区的语法简单。下面是使用哈希分区创建分区表的实例，如实例 6-36 所示。

【实例 6-36】创建 hash 分区。

```
CREATE TABLE sales_hash(
    Salesman_id number(5),
    Salesman_name varchar2(30),
    Sales amount number(10),
    Week_no number(2))
PARTITION BY HASH(salesman_id)
PARTITION 4
STORE IN (tbs1,tbs2,tbs3,tbs4)
```

上述实例创建了一个哈希分区，表名为 sales_hash，依据 salesman_id 创建哈希分区表，注意该分区表使用 4 个表空间来循环存储数据。

4. 复合分区

复合分区首先根据范围进行表分区，然后使用列表方式或者哈希方式创建子分区，显然使用复合分区实现了对分区表的更精细管理，既可以发挥范围分区的可管理优势，也可以发挥哈希分区的数据分布、条带化和并行化的优势。

【实例 6-37】创建一个复合分区表。

```
CREATE TABLE sales_composite (
  salesman_id NUMBER(5),
  salesman name VARCHAR2(30),
  sales_amount NUMBER(10),
  sales_date DATE)
PARTITION BY RANGE(sales_date)
SUBPARTITION BY HASH(salesman_id)
SUBPARTITION TEMPLATE(
SUBPARTITION sp1 TABLESPACE ts1,
SUBPARTITION sp2 TABLESPACE ts2,
SUBPARTITION sp3 TABLESPACE ts3,
SUBPARTITION sp4 TABLESPACE ts4)
(PARTITION sales_jan2008 VALUES LESS THAN(TO_DATE('02/01/2008','MM/DD/YYYY'))
PARTITION sales_feb2008 VALUES LESS THAN(TO_DATE('03/01/2008','MM/DD/YYYY'))
PARTITION sales_mar2008 VALUES LESS THAN(TO_DATE('04/01/2008','MM/DD/YYYY'))
PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2008','MM/DD/YYYY'))
PARTITION sales may2008 VALUES LESS THAN(TO DATE('06/01/2008','MM/DD/YYYY'))
);
```

在实例 6-37 中，我们创建了一个复合分区表，第一次分区使用范围分区，分区关键字为 sales_date，第二次分区即子分区，根据字段 salesman_id 创建哈希分区，子分区 sales_jan2008_sp1 存储在表空间 ts1 中，sales_feb2008_sp2 存储在表空间 ts2 中，其他分区表的存储空间依次类推。

6.6.2　分区表的优势

分区表的优势是把一个大表按照一定的规则分布在不同的分区块中，因为这样的表具有完全相同的逻辑结构，如相同的列名、列属性等。这样的表对用户透明，同时又带来一系列的好处。

- 用户可以在分区级完成数据加载、备份恢复等操作，大大减少了此类操作的执行时间。
- 使用分区表，Oracle 实现了一种分区剪裁技术，使用这种技术如果查询结果来自几个分区而不是整个表，则只查询需要的几个分区，这样就极大地提高了查询性能。
- 由于表分区的存在使得几个用户可以同时操作一个表的不同分区，实现了对“同一个表”的同步操作。
- 分区表与应用程序无关，如果在数据库设计时，一个表为非分区表，而由于业务的需要该表急剧增大，需要使用分区表，此时非分区表转化成分区表后，不用修改对该表的查询语句，如 select 等，就可以使用分区表。

6.7　本章小结

本章重点介绍了如何管理和维护表。数据类型是在表维护中非常重要的概念，因为表中存储的数据都必须定义数据类型（NULL 除外），这样在设计表时，选择对应的数据类型满足生产数据库中表的需要。创建表是必须掌握的基本内容，而表的维护也是必须熟练操作的内容，如表参数维护、维护表的列。在不需要一张表时，可以删除该表，此时使用 DROP 指令，不但会删除表中的数据，同时删除了表的结构，而删除的表是不可恢复的，所以在使用 DROP 指令时要确信不再需要该表。

分区表也是 Oracle 很有特色的部分，分区表把一个表按照一定的规则进行部署，分配到不同的磁盘上，这样使得一个表物理地分布在不同的地点，对于同步操作表和提高数据查询带来极大地性能提升，一般如果一个表增加到 2G 大小，就需要转换为分区表，分区表对应用程序是透明的，即普通表转换成分区表后，应用程序对原表的数据查询等操作不受影响。

第 7 章

数据字典

数据字典是在数据库创建时，由 Oracle 数据库服务器自动创建的一个额外的对象，这些对象存放在数据文件中，这些对象包括基表和数据字典视图，其中基表在 Oracle 数据库服务器中使用 CREATE DATABASE 创建，因为基表中的数据格式是无法直接阅读的，所以 Oracle 使用数据字典视图收集基表的信息，该数据字典视图是可读的，对 DBA 更有使用价值，数据字典视图通过 catalog.sql 脚本文件创建，那么数据字典中到底存储了哪些信息、如何使用和操作数据字典视图以及对 DBA 来讲有哪些常用的数据字典视图呢？我们将在接下来的小节中依次讲解。

7.1 数据字典中的内容

数据字典是很重要的数据库对象之一，它在数据库创建时由数据库服务器创建，记录了数据库创建的信息、各种对象的信息等，下面列出数据字典中包含的内容。

- 所有的模式（用户）对象的定义，这些对象包括表、视图、索引、簇、同义词、序列号、存储过程、函数触发器等。
- 数据库的逻辑结构和物理结构，如数据文件和重做日志文件的信息等。
- 所有模式对象被分配多少存储空间，以及当前使用的空间。
- 默认列的值。
- 对象完整性约束信息。
- 用户信息。
- 用户或角色的特权信息。
- 审计信息，如哪个用户具有访问或者修改某些模式对象的权利。

数据字典视图由 Oracle 数据库服务器自动创建并维护，也就是说只有 Oracle 服务器可以修改数据字典中的数据，在数据库运行期间，数据库结构或其他对象的变化信息会及时地记录在数据字典基表中，通过动态性能视图用户可以查看可读的数据字典基表中的信息。

7.2 数据字典视图的分类

数据字典视图分为三类，这些视图都是静态视图（静态的含义是这些视图在数据库运行期间不会发生变化，除非执行 ANALYZE 指令），这三类数据字典视图以不同的前缀区分彼此。数据字典名和对数据字典的解释如下所示。

- DBA_***：该视图包含数据库中整个对象的信息，以 DBA 为前缀的视图只能由管理员查询，不要在这些视图上创建同义词。
- ALL_***：该视图包含某个用户所能看到的全部数据库信息，包括当前用户所拥有的模式对象和用户可以访问的其他公共对象，还有通过授权或授予角色可以访问的模式对象。
- USER_***：该视图包含当前用户访问的数据库对象信息，它反映了数据库中某个用户的全部情况，该类视图隐含了 owner 信息，其全部内容为以 ALL 为前缀的视图的子集。



上述的“***”号表示数据库模式对象，如表 TABLE、索引 INDEX、视图 VIEW 对象、OBJECTS 等。

下面分别用具体的实例演示上面三种视图。

【实例 7-1】使用 DBA 用户连接数据库。

```
SQL> conn /as sysdba
已连接。
```

查看具有 DBA 前缀的视图时，会输出整个数据库的全局视图，但是这个视图只有具有 DBA 权限的用户才可以访问。如果某个用户具有 SELECT ANY TABLE 的权限，也可以查询具有 DBA 前缀的数据字典视图。在实例 7-1 中，我们成功登录数据库。使用实例 7-2 查看 dba_objects 视图的列定义。

【实例 7-2】查看 dba_objects 视图结构。

```
SQL> DESC dba_objects;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ----
 OWNER                                              VARCHAR2(30)
 OBJECT_NAME                                        VARCHAR2(128)
 SUBOBJECT_NAME                                     VARCHAR2(30)
 OBJECT_ID                                          NUMBER
 DATA_OBJECT_ID                                     NUMBER
 OBJECT_TYPE                                        VARCHAR2(18)
 CREATED                                            DATE
 LAST_DDL_TIME                                      DATE
 TIMESTAMP                                          VARCHAR2(19)
 STATUS                                             VARCHAR2(7)
 TEMPORARY                                          VARCHAR2(1)
```

```
GENERATED                                            VARCHAR2(1)
SECONDARY                                            VARCHAR2(1)
```

在确定了 dba_objects 视图中列的定义后，就可以使用实例 7-3 来查询数据字典视图 dba_objects 的内容，不过查询结果有大量输出，所以要采用一些限制条件来减少输出，并且为了使输出更易于阅读，需要事先做一些格式化工作。

【实例 7-3】通过数据字典视图 dba_objects 查看 SCOTT 用户的数据库对象信息。

```
SQL> col owner for a20
SQL> col object_name for a40
SQL> select owner,object name,created
  2  from dba_objects
  3* where owner = 'SCOTT'

OWNER                OBJECT_NAME                            CREATED
---------------------------------- -------------------------------------
SCOTT                BONUS                                  12-5 月 -09
SCOTT                DEPT                                   04-9 月 -01
SCOTT                DEPT_TEMP                              08-10 月-08
SCOTT                EMP                                    12-5 月 -09
SCOTT                EMP_TEMP                               27-9 月 -08
SCOTT                ORD                                    11-5 月 -09
SCOTT                ORD_ORDNO                              11-5 月 -09
SCOTT                PK_DEPT                                04-9 月 -01
SCOTT                PK_EMP                                 12-5 月 -09
SCOTT                PRODUCT                                04-10 月-08
SCOTT                SALGRADE                               12-5 月 -09

OWNER                OBJECT_NAME                            CREATED
---------------------------------- --------------------------
SCOTT                SUPPLIER                               10-5 月 -09

已选择 12 行。
```

当然，上述查询使用 ALL_OBJECTS 视图也可以实现，DBA 用户可以访问所有的数据库信息。为了查看 all_objects 的信息，我们先查看该视图的列定义，如实例 7-4 所示。

【实例 7-4】查看 all_objects 结构信息。

```
SQL> DESC all_objects;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- -----------------
 OWNER                                     NOT NULL VARCHAR2(30)
 OBJECT NAME                               NOT NULL VARCHAR2(30)
 SUBOBJECT_NAME                                     VARCHAR2(30)
 OBJECT_ID                                 NOT NULL NUMBER
 DATA_OBJECT_ID                                     NUMBER
 OBJECT_TYPE                                        VARCHAR2(18)
 CREATED                                   NOT NULL DATE
 LAST_DDL_TIME                             NOT NULL DATE
```

```
TIMESTAMP                                            VARCHAR2(19)
STATUS                                               VARCHAR2(7)
TEMPORARY                                            VARCHAR2(1)
GENERATED                                            VARCHAR2(1)
SECONDARY                                            VARCHAR2(1)
```

观察实例 7-2 和实例 7-4 会发现二者输出是一样的，也就是说“DBA_***”视图，和“ALL_***”视图具有相同的列定义。

我们使用 SCOTT 用户登录数据库，如实例 7-5 所示。

【实例 7-5】使用 SCOTT 用户登录数据库。

```
SQL> conn scott/tiger;
已连接。
```

我们先使用实例 7-6 查询当前 SCOTT 用户的 all_objects 表中有多少个 owner，名字是什么。

【实例 7-6】当前 SCOTT 用户的 all_objects 表中 owner 的名字。

```
SQL> select distinct(owner)
  2  from all_objects;

OWNER
--------------------
CTXSYS
MDSYS
OLAPSYS
ORDPLUGINS
ORDSYS
PUBLIC
SCOTT
SYS
SYSTEM
WKSYS

已选择 10 行。
```

我们发现数据字典视图 all_objects 中共有 10 个 owner，其中一个 owner 为 SCOTT，虽然我们使用 SCOTT 用户登录，但是该用户可以访问其他用户的部分对象信息。为了减少输出，我们用实例 7-7 说明如何查询 all_objects 表中特定 owner 的信息。

【实例 7-7】查询 all_objects 表中特定 owner 的信息。

```
SQL> select owner,object_name,created
  2  from all_objects
  3* where owner = 'SCOTT'

OWNER                OBJECT_NAME                           CREATED
-------------------- ---------------------------------------
SCOTT                BONUS                                12-5 月 -09
SCOTT                DEPT                                  04-9 月 -01
```

```
SCOTT                DEPT_TEMP                                  08-10月-08
SCOTT                EMP                                        12-5月 -09
SCOTT                EMP_TEMP                                   27-9月 -08
SCOTT                ORD                                        11-5月 -09
SCOTT                ORD_ORDNO                                  11-5月 -09
SCOTT                PK_DEPT                                    04-9月 -01
SCOTT                PK_EMP                                     12-5月 -09
SCOTT                PRODUCT                                    04-10月-08
SCOTT                SALGRADE                                   12-5月 -09

OWNER                OBJECT_NAME                                 CREATED
-------------------- --------------------------------------------------
SCOTT                SUPPLIER                                   10-5月 -09

已选择 12 行。
```

输出结果说明已成功查询 all_objects 视图中用户 SCOTT 的信息，包括 SCOTT 用户下的对象名和对象创建时间。

如果在 SCOTT 用户模式下查询 dba_objects 视图，就会出错，因为该用户不具备 DBA 权限，也没有 SELECT ANY TABLE 的权限。查询结果如实例 7-8 所示。

【实例 7-8】在 SCOTT 用户模式下查询 dba_objects 视图。

```
SQL> select owner,object_name,created
  2  from dba_objects
  3  where owner = 'SCOTT';
from dba objects
     *
ERROR 位于第 2 行:
ORA-00942: 表或视图不存在
```

下面演示如何查看“USER_***”数据字典视图，此时还是使用 SCOTT 用户登录数据库系统。查看 user_objects 的定义，如实例 7-9 所示。

【实例 7-9】查看静态数据字典 user_objects 的定义。

```
SQL> desc user_objects;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ------------------
 OBJECT_NAME                                        VARCHAR2(128)
 SUBOBJECT_NAME                                     VARCHAR2(30)
 OBJECT ID                                          NUMBER
 DATA_OBJECT_ID                                     NUMBER
 OBJECT_TYPE                                        VARCHAR2(18)
 CREATED                                            DATE
 LAST DDL TIME                                      DATE
 TIMESTAMP                                          VARCHAR2(19)
 STATUS                                             VARCHAR2(7)
 TEMPORARY                                          VARCHAR2(1)
 GENERATED                                          VARCHAR2(1)
 SECONDARY                                          VARCHAR2(1)
```

比较实例 7-2、实例 7-4 和实例 7-9 可以发现实例 7-9 中视图 user_objects 的列没有 OWNER。其实，这个问题也容易理解，因为“USER_***”视图是当前用户的数据库对象信息，既然是当前用户自然不需要有 OWNER 选项了，而“DBA_***”视图和“ALL_***”视图，包括不同用户的对象信息，使用 OWNER 列可以区别不同用户的对象信息。我们使用实例 7-10 查询 SCOTT 用户的对象信息。

【实例 7-10】查询 SCOTT 用户的对象信息。

```
SQL> select object_name,object_type,created
  2  from user objects;

OBJECT_NAME                                    OBJECT_TYPE        CREATED
----------------------------------------------------------------------
BONUS                                         TABLE               12-5 月 -09
DEPT                                          TABLE               04-9 月 -01
DEPT_TEMP                                     TABLE               08-10 月-08
EMP                                           TABLE               12-5 月 -09
EMP_TEMP                                      TABLE               27-9 月 -08
ORD                                           TABLE               11-5 月 -09
ORD_ORDNO                                   SEQUENCE            11-5 月 -09
PK_DEPT                                       INDEX               04-9 月 -01
PK_EMP                                        INDEX               12-5 月 -09
PRODUCT                                       TABLE               04-10 月-08
SALGRADE                                      TABLE               12-5 月 -09

OBJECT_NAME                                    OBJECT_TYPE        CREATED
-------------------------------------------------------------------
SUPPLIER                                      TABLE               10-5 月 -09

已选择 12 行。
```

在该实例中，我们查询了当前用户的对象名 OBJECT_NAME、对象类型 OBJECT_TYPE 和对象创建时间 CREATED，其实，这个输出结果中的对象名和实例 7-7 相同。

7.3　数据字典视图的使用

数据字典视图是静态视图，在数据库重新启动前，静态数据字典中的信息是不会变化的。有一些数据字典视图对于 DBA 而言很重要，下面依次介绍这些数据字典视图。

1. user_tables 视图

该视图可以查看当前用户所有的表，如实例 7-11 所示。

【实例 7-11】查看当前用户所有的表。

```
SQL> conn scott/tiger
已连接。
```

```
SQL> select table_name
  2  from user_tables;

TABLE_NAME
------------------------------
BONUS
DEPT
EMP
SALGRADE

已选择 8 行。
```

2. user_indexes 数据字典视图

用于查看当前用户创建的索引，索引在某种程度上可以加快查询的速度，如实例 7-12 所示。

【实例 7-12】查看当前用户创建的索引。

```
SQL> select index_name
  2  from user_indexes;

INDEX_NAME
------------------------------
PK_DEPT
PK_EMP
SYS_C003123
SYS_C003125
SYS_C003129
SYS_C003130

已选择 6 行。
```

3. user_views 数据字典视图

用于查看当前用户拥有的视图，如实例 7-13 所示。

【实例 7-13】查看当前用户拥有的视图。

```
SQL> select view_name
  2* from user_views

VIEW_NAME
------------------------------
STUDENT_SCORE
VIEW_DEPT
```

4. user_catalog 视图

该视图包含当前用户的所有表的名字和类型。

【实例 7-14】查询该视图的结构。

```
SQL> desc user_catalog;
 名称                                    是否为空？ 类型
```

```
 ----------------------------------------- --------- ---------------------
 TABLE_NAME                                 NOT NULL VARCHAR2(30)
 TABLE_TYPE                                          VARCHAR2(11)
```

输出结果说明，该视图有两列，一个为 TABLE_NAME，该列不能为空，是变长字符类型，另一个为 TABLE_TYPE，该列可以为空，是变长字符类型，如实例 7-15 所示。

【实例 7-15】显示用户 SCOTT 的所有表名和类型。

```
SQL> select *
  2  from user_catalog;

TABLE_NAME                     TABLE_TYPE
----------------------------------------------------------------------
BONUS                          TABLE
DEPT                           TABLE
DEPT_TEMP                      TABLE
EMP                            TABLE
EMP_TEMP                       TABLE
ORD                            TABLE
ORD_ORDNO                      SEQUENCE
PRODUCT                        TABLE
SALGRADE                       TABLE
SUPPLIER                       TABLE

已选择 10 行。
```

5. dba_users 视图

用于查看数据库系统上何时创建了多少个用户，如实例 7-16 所示。

【实例 7-16】查看数据库系统上创建的用户信息。

```
SQL> select username,created
  2  from dba_users;

USERNAME                       CREATED
------------------------------------------------ ------------------
SYS                            04-9月 -01
SYSTEM                         04-9月 -01
DBSNMP                         04-9月 -01
AURORA$JIS$UTILITY$            04-9月 -01
AURORA$ORB$UNAUTHENTICATED     04-9月 -01
SCOTT                          04-9月 -01
…………
```

7.4　动态性能视图的使用

Oracle 还维护了另一类非常重要的数据字典视图：动态性能视图。动态性能视图只存在于运

行的数据库中，它是一组虚表，通常也把这组表称为动态性能表（Dynamic Performance Table）。

数据库的动态性能视图只有管理员用户可以查询，而其他普通用户不需要查询这些虚表中的信息。管理员可以在动态性能视图上创建视图，并将访问权限授予其他用户。任何用户都无法修改或删除动态性能视图，所以有时这些动态性能视图也被称为固定视图（Fixed View）。

SYS 用户拥有所有的动态性能视图，这些动态性能视图以“v$”为前缀，如 v$controlfile 包含了控制文件存储目录和文件名信息，v$datafile 包含了数据库文件信息，v$fixed_table 视图包含了当前所有动态性能视图。

如果用户想知道当前运行的数据库中所有的动态性能视图，可以使用 v$fixed_table 实现，不过一般该视图会输出大量的记录，不方便阅读，最好使用 spool 工具存储输出结果，再分析存储的输出信息。

为了更充分地理解动态性能视图的作用和使用，我们给出如下的实例进行说明，这些实例也是在实际工作中经常使用的。

【实例 7-17】查询和日志文件相关的信息。

```
SQL> conn /as sysdba
已连接。
SQL> select *
  2  from v$fixed_table
  3  where name like 'V$LOG%';

NAME                                   OBJECT_ID TYPE   TABLE_NUM
------------------------------------------------------------------------
V$LOGFILE                              4294950935 VIEW      65537
V$LOG                                  4294951049 VIEW      65537
V$LOGHIST                              4294951051 VIEW      65537
V$LOG_HISTORY                          4294951077 VIEW      65537
V$LOGMNR_CONTENTS                      4294951541 VIEW      65537
V$LOGMNR_LOGS                          4294951543 VIEW      65537
V$LOGMNR_DICTIONARY                    4294951545 VIEW      65537
V$LOGMNR_PARAMETERS                    4294951547 VIEW      65537
V$LOGMNR_LOGFILE                       4294951643 VIEW      65537
V$LOGMNR_PROCESS                       4294951646 VIEW      65537
V$LOGMNR_TRANSACTION                   4294951649 VIEW      65537

NAME                                   OBJECT_ID TYPE   TABLE_NUM
-------------------------------------------------------------------------
V$LOGMNR_REGION                        4294951633 VIEW      65537
V$LOGMNR_CALLBACK                      4294951636 VIEW      65537
V$LOGMNR_SESSION                       4294951640 VIEW      65537
V$LOGSTDBY_COORDINATOR                 4294951655 VIEW      65537
V$LOGSTDBY_APPLY                       4294951658 VIEW      65537
V$LOGSTDBY                             4294951711 VIEW      65537
V$LOGSTDBY_STATS                       4294951714 VIEW      65537

已选择 18 行。
```

该实例中查询了所有和日志文件相关的动态性能视图，如果想了解日志文件的详细信息我们可以使用 v$log 视图和 v$logfile 视图，如实例 7-18 和实例 7-19 所示。

【实例 7-18】查看日志组的状态信息。

```
SQL> select group#,members,archived,status
  2  from v$log;

    GROUP#    MEMBERS ARC STATUS
---------- ---------- --- ----------------
         1          1 NO  CURRENT
         2          1 NO  INACTIVE
         3          1 NO  INACTIVE
```

该实例的作用是查看当前正在使用的重做日志组，STATUS 为 CURRENT 说明该日志组正在使用中，STATUS 为 INACTIVE 说明当前数据库系统没有使用该重做日志组。上例说明当前有三个重做日志组，第一个日志组正在使用中。

【实例 7-19】查看重做日志文件信息。

```
SQL> col member for a40
SQL> select *
  2* from v$logfile

    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- ----------------------------------------
         3 STALE   ONLINE  C:\ORACLE\ORADATA\LIN\REDO03.LOG
         2 STALE   ONLINE  C:\ORACLE\ORADATA\LIN\REDO02.LOG
         1         ONLINE  C:\ORACLE\ORADATA\LIN\REDO01.LOG
```

视图 v$logfile 用于查看当前数据库系统的重做日志组的日志成员的存储目录、文件名和状态信息。该输出说明当前的日志组 1 正在使用中，当前的日志组中日志成员的存储目录和文件名为 C:\ORACLE\ORADATA\LIN\REDO01.LOG。

【实例 7-20】查询当前正在使用的重做日志文件的信息。

```
SQL> select l.group#,l.archived,l.status,lf.type,lf.member
  2  from v$log l,v$logfile lf
  3  where l.group# = lf.group#
  4  and l.status = 'CURRENT';

GROUP# ARC STATUS           TYPE    MEMBER
------------ --- ------------------------------ ------------- --
     1 NO  CURRENT          ONLINE  C:\ORACLE\ORADATA\LIN\REDO01.LOG
```

从 v$log 视图和 v$logfile 视图的联合查询可以看出，当前数据库正在使用的日志文件组为 group1，数据库运行在非归档模式，该日志组有一个日志成员，存储目录为 C:\ORACLE\ORADATA\BJYZZ，文件名为 REDO01.LOG。

【实例 7-21】通过 v$instance 视图查看实例信息。

```
SQL> col instance_name for a20
SQL> col host_name for a10
SQL> select instance_name,host_name,version,startup_time,logins
  2* from v$instance

INSTANCE_N HOST_NAME           VERSION          STARTUP_TI LOGINS
------------------- ------------------------------------ ------
lin        LINSHUZE-6F370C          9.0.1.1.1         16-6 月 -09   ALLOWED
```

上述输出说明，当前的实例名为 lin，主机名为 LINSHUZE-6F370C，版本号为 9.0.1.1.1，实例的启动时间为 16-6 月-09。

【实例 7-22】查看当前数据库的信息。

```
SQL> col name for a10
SQL> select name,created,log mode
  2  from v$database;

NAME       CREATED    LOG_MODE
---------- ---------- ------------
LIN        13-5 月 -09 ARCHIVELOG
```

输出结果说明，该数据库名字为 LIN，数据库创建时间为 13-5 月 -09，该数据库运行在非归档模式。

总之，动态性能视图很好地反映了当前数据库的运行状态信息，对于数据库性能优化和判断系统瓶颈提供信息支持，通过动态性能视图还可以查看控制文件的信息、数据文件的信息和表空间信息等，DBA 用户经常使用的动态性能视图，这里不再一一介绍，在以后的章节中会继续使用各种动态性能视图，本节只是对读者起到引导作用，希望读者理解动态性能视图的概念和应用。

7.5　本章小结

数据字典是数据库中非常重要的数据库对象，它在创建数据库时创建，其中记录了数据库创建信息和各种对象的信息。由于基表的内容无法阅读，所以 Oracle 提供了数据字典视图，该视图是基于基表数据的。数据字典视图是静态视图，在数据库启动后就无法改变。Oracle 数据库还维护了动态性能视图，这些视图以 v$为前缀，动态性能视图反映了数据库当前的运行状态和各种对象的活跃信息，动态性能视图是 DBA 进行故障判断和数据库性能调优的重要依据。

第 8 章

◀视　　图▶

本章主要讲解普通视图和物化视图。普通视图是一个虚表，不占用存储空间，在数据字典中只有视图的定义，视图可以通过 DML 语言操作，但是有一定限制，因为操作视图最终还是操纵创建视图的底层表。物化视图是 Oracle 10g 中提出的概念，物化视图是一种占用存储空间的特殊视图，物化视图用于完成数据复制、数据同步、数据汇总以及数据发布，物化视图在生产数据库中很有实用价值，尤其在分布式计算环境和移动计算环境中，我们将通过介绍物化视图的概念、如何创建物化视图以及在使用物化视图中遇到的一些问题详细介绍物化视图。

8.1　什么是视图

首先视图是一种虚表，它不存储数据，在 Oracle 的数据字典中只是记录了视图的定义，视图通过 select 语句定义。

在多表查询中为了简化查询过程，创建基于多表的一个查询视图，用户每次通过查询视图类查询所需要的数据。用户可以查询视图甚至使用 UPDATE、DELETE 或者 INSERT 语句操纵视图，但是此时操纵的还是基表中的数据，切记：视图只有定义没有物理存储，在操作视图时实际上是通过执行 SQL 语句操纵定义视图的实际的物理表。

视图的优点如下。

- 减少数据操纵的复杂性：在执行基于多表的查询中，用户需要输入一长串的列名、使用表的别名、输入复杂的条件语句等，显然如果每次输入这样复杂的操作对于非专业人员是件“痛苦”的事，而使用视图，则可以简化这些语句输入，并且用一个易于记忆的名字命名视图。
- 增强安全性： 可以将视图设置为 READ ONLY，这样使用者就无法修改或更新数据，并且相同的数据可以显示在不同的视图中，如果一个表数据很重要，但是用户需要访问该表中的某列值，则可以使用视图查询该重要表中的该列值，而不用查询这个表。
- 重命名列：很多表的列名是开发数据库的专业人员使用的，而对于非专业人员理解起来不直观，通过创建视图、重命名列表达列名含义，使其具有更清晰直观的效果。
- 实现数据定制：本章创建的视图是基于部门创建的员工表，这样一个部门内部的员工通过授权就可以只看到自己部门内部的员工信息，实现了定制的本单位员工数据。
- 保护数据的完整性：通过视图的 WITH CHECK OPTION 子句实现数据行的完整性约束

和数据有效性检查。

8.2　创建视图

下面通过实例来体验和说明如何创建视图。我们使用 SCOTT 用户的表来创建视图，但是必须授予 SCOTT 用户创建视图的权限，授权方法如实例 8-1 所示。

【实例 8-1】授予 SCOTT 用户创建视图的权限。

```
SQL> conn system/oracle as sysdba
已连接。
SQL> grant create view to scott;
```

在 SCOTT 用户模式下，有一表对象 EMP，即员工表，该表记录了员工号、员工名字、工作性质、雇佣时间、薪水以及部门号等。为了方便每个部门查询自己部门内部的员工信息，我们为每个部门创建一个视图，这样不同的部门只要使用视图就可以完成查询，而不用再使用多表连接和 WHERE 条件语句来限制查询的部门（虽然实际的操作还是一样“复杂”，但至少对使用者简单），并且通过和表 DEPT 的联合查询给出该部门的名字。创建属于 ACCOUNTING 部门的员工视图，如实例 8-2 所示。

【实例 8-2】创建属于 ACCOUNTING 部门的员工视图。

```
SQL>  create view accounting view as
  2  select e.ename "employee_name",e.job "job",e.hiredate "hiredate",
     e.sal "salary",d.dname "dep_name"
  3  from dept d,emp e
  4  where e.deptno =d.deptno
  5 and d.deptno <20;

视图已创建。
```

从上述创建视图的实例可以看出，使用 CREATE VIEW viewname AS 语句创建视图，AS 后是 SQL 查询语句，一旦视图创建成功，则在数据字典中会记录该视图的定义，如实例 8-3 所示，用于查询数据字典中记录的视图定义。

【实例 8-3】查询数据字典中记录的视图定义。

```
SQL> select view_name
  2 from user views;

VIEW_NAME
--------------------
ACCOUNTING_VIEW
```

我们再查询该视图的定义语句，在视图 USER_VIEWS 中使用 TEXT 属性列记录该视图的定义，如实例 8-4 所示。

【实例 8-4】查询视图 ACCOUNTING_VIEW 的定义。

```
SQL> select text
  2  from user_views
  3  where view_name = 'ACCOUNTING_VIEW';

TEXT
--------------------------------------------------------------------------
select e.ename "employee_name",e.job "job",e.hiredate "hiredate",e.sal
"salary",
```

创建视图后，再需要查询关于 ACCOUNTING 部门的员工信息时，就可以通过视图 ACCOUNTING_VIEW 来实现，并且该视图的列对基表进行了重命名。查询 ACCOUNTING 部门的所有员工信息，如实例 8-5 所示。

【实例 8-5】查询 ACCOUNTING 部门的所有员工信息。

```
SQL> select *
  2  from accounting_view;

employee_n job          hiredate          salary dep_name
--------------------------------- -----------------------------------
CLARK   MANAGER   09-6 月 -81      2450 ACCOUNTING
KING     PRESIDENT  17-11 月-81      5000 ACCOUNTING
MILLER   CLERK      23-1 月 -82      1300 ACCOUNTING
```

我们也可以通过以下方式创建视图，如此时创建部门 SALES 的员工表，如实例 8-6 所示。

【实例 8-6】创建部门 SALES 的员工视图。

```
SQL>  create or replace view sales_view
  2  ("employee_name","job","hiredate","salary","dep_name")
  3  as
  4  select e.ename,e.job,e.hiredate,e.sal,d.dname
  5  from dept d,emp e
  6  where e.deptno = d.deptno
  7  and d.deptno = 30;

视图已创建。
```

查询是否成功创建 SALES_VIEW 视图，如实例 8-7 所示。

【实例 8-7】查询是否成功创建 SALES_VIEW 视图。

```
SQL> select view name,text
  2  from user_views
  3* where view_name like 'SAL%'

VIEW_NAME
------------------------------
TEXT
------------------------------------------------
```

```
SALES_VIEW
select e.ename,e.job,e.hiredate,e.sal,d.dname
from dept d,emp e
where e.deptno =
```

从实例 8-7 的输出可以看出，已成功创建视图 SALES_VIEW。

在实例 8-6 中，我们使用了 replace 参数，并且将别名放在了视图名字的后面，这样做是允许的。虽然与实例 8-2 的创建视图方式不同，但都是正确的创建方式。当需要查询部门 SALES 的员工信息时，就可以直接使用 SALES_VIEW 视图，而不必使用复杂的查询语句，如实例 8-8 所示。

【实例 8-8】使用视图的查询。

```
SQL> select *
  2  from sales_view;

employee n    job          hiredate      salary, dep name
------------------ -------------------- ----------------------------
ALLEN      SALESMAN  20-2 月 -81     1600  SALES
WARD       SALESMAN  22-2 月 -81     1250  SALES
MARTIN     SALESMAN  28-9 月 -81     1250  SALES
BLAKE      MANAGER   01-5 月 -81     2850  SALES
TURNER     SALESMAN  08-9 月 -81     1500  SALES
JAMES      CLERK       03-12 月-81      950  SALES

已选择 6 行。
```

下面总结创建视图的语法格式，如下所示。

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view_name
[别名[,别名]......]
AS
查询子句
[WITH CHECK OPTION [CONSTRAINT 约束名]]
[WITH READ ONLY]
```

下面依次介绍每个选项。

- CREATE OR REPLACE：创建视图，如果所创建的视图名存在，则用新创建的视图覆盖原视图。
- FORCE/NOFORCE：FORCE 说明不论创建视图时，基表是否存在，都创建该视图，NOFORCE 则相反，只有所引用的基表都存在时才创建该视图。
- 别名：就是所创建的视图的列名，数量与视图所产生的列的数量相等。
- AS：该关键字说明下面是查询子句，用于定义视图。
- 查询子句：是任意正确的、完整的查询语句。
- WITH CHECK OPTION：当更新某一数据行时，必须满足 WHERE 子句的条件。
- WITH READ ONLY：设置该视图为“只读”状态，说明无法在该视图上进行任何 DML 操作。

8.3　使用视图

在创建视图的语法格式中，我们给出了完整的语法格式，注意有两个 WITH 子句的使用，使得使用视图时更加安全。因为视图更多的是给非专业人员使用的，所以要考虑使用者可能的操作。

试想你创建了一个视图，而视图的使用者可以随意更改视图、修改某一列的值等，显然这样的操作会直接反映在创建这个视图的基表上，使得基表数据被改变，这是不合理的。而使用 WITH READ ONLY 子句就可以很好地解决找个问题，而 WITH CHECK OPTION 可以增加约束条件，使得用户对数据的更新受到某些限制。下面依次讲解如何使用 WITH READ ONLY 子句和 WITH CHECK OPTION 子句。

1. WITH READ ONLY 子句

创建部门 RESEARCH 的员工信息，如实例 8-9 所示。

【实例 8-9】创建部门 RESEARCH 的员工视图。

```
SQL>  create or replace view research_view
  2   ("employee_name","job","hiredate","salary","dep_name")
  3   as
  4   select e.ename,e.job,e.hiredate,e.sal  ,d.dname
  5   from dept d,emp e
  6   where e.deptno = d.deptno
  7   and d.deptno = 20
  8   with read only;

视图已创建。
```

在该实例中，我们使用了 WITH READ ONLY 子句，也就是不允许对该视图使用 DML 操作，否则提示如实例 8-10 所示的错误。

【实例 8-10】对该视图 research_view 使用 DML 操作的错误提示。

```
SQL> update research_view
  2  set "salary" = 1000
  3* where  "job" = 'CLERK'
set "salary," = 1000
    *
第 2 行出现错误:
ORA-01733: 此处不允许虚拟列
```

再使用 DELETE 语句测试是否可以删除数据，如实例 8-11 所示。

【实例 8-11】测试是否可以删除数据。

```
SQL> delete from research_view
  2  where "job" = 'CLERK';
delete from research_view
            *
```

```
第 1 行出现错误:
ORA-01752: 不能从没有一个键值保存表的视图中删除
```

实例 8-10 和实例 8-11 说明通过 WITH READ ONLY 子句创建的视图不能使用 DML 语言。

2. WITH CHECK OPTION 子句

使用该子句表明当通过视图更新数据或删除数据时不能违背 WHERE 子句限制的条件。如先创建一个视图，该视图只涉及一个表，如实例 8-12 所示。

【实例 8-12】创建一个基于单表的视图。

```
SQL> create view emp view as
  2  select *
  3  from emp
  4  where JOB IN ('SALESMAN','MANAGER');

视图已创建。
```

实例 8-12 的 WHERE 子句限制只选择表 EMP 中 JOB 为 SALESMAN 和 MANAGER 的所有员工数据，即只有 JOB 为 SALESMAN 和 MANAGER 的数据才可以插入，否则不允许插入。

下面，我们试图插入一行员工数据，且 JOB 为 Marketing，如实例 8-13 所示。

【实例 8-13】尝试向视图 emp_view 中插入一行员工数据。

```
SQL> insert into emp view(empno,ename,job,mgr,hiredate,
  2  sal,comm,deptno)
  3  values (7565,'TOM','Marketing',7998,SYSDATE,2000,22,40);
insert into emp_view(empno,ename,job,mgr,hiredate,
                  *
第 1 行出现错误:
ORA-01402: 视图 WITH CHECK OPTIDN where 子句违规
```

此时，不允许插入该行数据，因为插入的数据违反了 WHERE 子句中 JOB IN ('SALESMAN','MANAGER')的条件。

【实例 8-14】向视图 emp_view 中插入一行员工数据。

```
SQL> insert into emp_view(empno,ename,job,mgr,hiredate,
  2  sal,comm,deptno)
  3* values (7565,'TOM','MANAGER',7839,SYSDATE,2000,22,20)

已创建 1 行。
```

显然实例 8-14 中的插入操作没有违反 WHERE 子句的约束，因为 JOB 为 MANAGER。为了确认插入结果可使用实例 8-15 验证结果。

【实例 8-15】确认实例 8-14 的插入结果。

```
SQL> select *
  2  from emp_view
  3  ename = 'TOM';
```

```
EMPNO ENAME   JOB        MGR  HIREDATE        SAL   COMM  DEPTNO
---------- ---------- --------- ---------- ---------- ----------
7565      TOM    MANAGER  7839  21-7月 -09        2000    22    20
```

我们知道，操纵视图最终是通过视图的定义来操作实际的表，所以实例 8-15 的插入操作实际上会在表 EMP 中插入该行数据，我们用实例 8-16 来验证。

【实例 8-16】查询表 EMP 中是否插入了一行数据。

```
SQL> select *
  2  from emp
  3  ename = 'TOM';
EMPNO ENAME   JOB        MGR  HIREDATE        SAL   COMM  DEPTNO
---------- ---------- --------- ---------- ---------- ------
7565      TOM    MANAGER  7839  21-7月 -09        2000    22    20
```

8.4　修改视图

如果需要修改视图的定义，此时 Oracle 提供的唯一方法就是重新定义该视图，用新定义的视图覆盖原来的视图，利用 CREATE OR REPLACE VIEW 子句实现。

在实例 8-9 中，我们创建了视图 RESEARCH_VIEW，该视图中包含了 RESEARCH 部门的所有员工信息，但是此时需要增加一个员工号 empno。首先，我们通过实例 8-17 查看视图 RESEARCH_VIEW 是否存在。

【实例 8-17】查看视图 RESEARCH_VIEW 是否存在。

```
SQL> select view_name
  2  from user_views;

VIEW_NAME
--------------------------
SALES_VIEW
RESEARCH VIEW
ACCOUNTING_VIEW
```

【实例 8-18】确认要查看的视图 research_view 的结构。

```
SQL> desc research view;
 名称                                    是否为空? 类型
 ----------------------------------------- --------- --------------
 employee_name                                      VARCHAR2(10)
 job                                              VARCHAR2(9)
 hiredate                                          DATE
 salary,                                           NUMBER(7,2)
 dep_name                                          VARCHAR2(14)
```

视图 RESEARCH_VIEW 有 5 个列属性，但是没有员工号 EMPNO，现在，我们修改视图

RESEARCH_VIEW，增加员工号属性并且将所有的列名改为中文，如实例 8-19 所示。

【实例 8-19】修改视图 RESEARCH_VIEW。

```
SQL> create or replace view research_view
  2  ("员工号","员工姓名","岗位","雇佣时间","薪水","部门")
  3  as
  4  select e.empno,e.ename,e.job,e.hiredate,e.sal,d.dname
  5  from dept d , emp e
  6  where e.deptno = d.deptno
  7* and d.deptno = 20

视图已创建。
```

查询该视图的列名信息，查看是否为当前我们创建的新视图的列名，如实例 8-20 所示。

【实例 8-20】确认是否创建新视图。

```
SQL> desc research_view;
 名称                                          是否为空? 类型
 ---------------------------------------------------------------
 员工号                                        NOT NULL NUMBER(4)
 员工姓名                                               VARCHAR2(10)
 岗位                                                   VARCHAR2(9)
 雇佣时间                                               DATE
 薪水                                                   NUMBER(7,2)
 部门                                                   VARCHAR2(14)
```

显然，旧的视图已经被修改，下面，我们查询该视图的一些数据，查看语句的输出效果，中文的列名对于用户来讲或许更友好，如实例 8-21 所示。

【实例 8-21】查询视图 research_view 的信息。

```
SQL> select *
  2  from research_view
  3  where rownum<3;

    员工号 员工姓名    岗位          雇佣时间              薪水  部门
---------------- ---------------- -------------- -----
      7565 TOM        MANAGER   21-7月 -09        2000  RESEARCH
      7369 SMITH      CLERK      17-12月-80        800   RESEARCH
```

8.5　管理视图

视图只是一个虚表，它不存储数据，对视图的操作最终是通过视图的定义操纵它所涉及的表，创建视图成功后，在 Oracle 数据库的数据字典中记录该视图的信息。Oracle 使用数据字典管理视图，该数据字典是 USER_VIEWS。

8.5.1 通过数据字典查询视图

因为在第 8.2 节中已经使用了数据字典 USER_VIEWS，所以这里只给出一个实例来查询当前我们已经创建的视图信息，如实例 8-22 所示。

【实例 8-22】查询当前我们已经创建的视图信息。

```
SQL> select view_name
  2  from user_views;

VIEW NAME
------------------------------
SALES_VIEW
RESEARCH VIEW
ACCOUNTING_VIEW
EMP_VIEW
```

可以看出，我们已经创建了 4 个视图，其中前三个视图是基于 EMP 和 DEPT 而创建的每个部门的员工信息视图，最后一个视图 EMP_VIEW 是使用单表 EMP 创建的视图。

8.5.2 Oracle 视图查询的内部过程

Oracle 在使用视图查询数据时，会执行一系列的分析过程，最终才执行 SQL 查询语句。操作过程如下：

- 读取数据字典，获得该视图的定义，查找到该视图所引用的表。
- 从数据字典中查询当前用户对于该视图所引用的表的权限。
- 执行定义该视图的 SQL 语句，实现执行视图查询。

为了更形象地理解视图查询的内部过程，给出如图 8-1 所示的过程图。

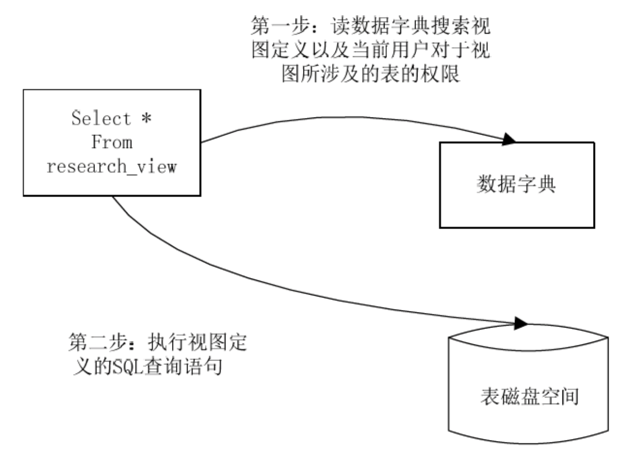


图 8-1 视图查询内部过程图



数据字典和表都保存在磁盘上，所以上述过程实际上要实现两次磁盘 I/O，这对系统的效率会有一定影响。

8.6 视图 DML 操作的限制

视图的 DML 操作是有限制的，毕竟视图的操作将最终转化成对它引用的表的物理操作，所以依据视图的类型不同、是否使用函数等条件，对于视图的 DML 操作具有一定的限制，下面分别介绍简单视图和复杂视图在 DML 操作中的限制。

1. 简单视图

简单视图从一个表读取数据，不包括函数和分组数据。简单视图可以进行 DML 操作，即可以对简单视图进行 DELETE、UPDATE 和 INSERT 操作，简单视图的 DML 操作直接转化成对定义它的表的 DML 操作。

2. 复杂视图

复杂视图从多个表提取数据，包括函数和分组数据，复杂视图不一定能进行 DML 操作。Oracle 对于在复杂视图上进行 DML 操作添加了很多限制条件，即：

- 如果复杂视图包含了分组函数、GROUP BY 子句或者 DISTINCT 关键字，就不能使用 DELETE、UPDATE 或 INSERT 的 DML 操作。在复杂视图上进行 DML 操作最终也要转化成对视图所引用表的 DML 操作，由于复杂视图中包含函数或分组函数，所以就不能在复杂视图上使用 DML 操作。
- 如果复杂视图中的列包含表达式，或者有伪列 ROWNUM，则不能使用复杂视图进行 UPDATE 或 INSERT 等 DML 操作。

8.7 删除视图

如果不需要一个视图，可以删除该视图，删除视图的指令是 DROP VIEW。删除视图 EMP_VIEW 如实例 8-23 所示。

【实例 8-23】删除视图 EMP_VIEW。

```
SQL> drop view emp_view;

视图已删除。
```

一旦删除了视图，要使用数据字典 USER_VIEWS 验证是否删除，如实例 8-24 所示。

【实例 8-24】使用数据字典 USER_VIEWS 验证是否删除。

```
SQL> select view_name
  2  from user_views;

VIEW NAME
----------------------------
SALES_VIEW
RESEARCH_VIEW
ACCOUNTING_VIEW
```

显然该实例中没有了视图 EMP_VIEW 的定义，说明删除成功。

8.8　物化视图

物化视图就是具有物理存储的特殊视图，占据物理空间。物化视图是基于表、物化视图等创建的。它需要和源表进行同步，不断地刷新物化视图中的数据，本节会讲到所有这些问题，并且使得读者掌握创建物化视图的条件，以及如何创建物化视图。

8.8.1　什么是物化视图

在以上几节中，我们重点讲解了视图的概念、使用和管理等，这些视图称为普通视图。在 Oracle 使用普通视图时，它会重新执行创建视图的所有 SQL 语句，如果这样的视图有多张表的 JION 或 ORDER BY 子句，而且表相当大，则会相当耗时。使用普通视图的查询效率很低。为了解决这个问题，Oracle 提出了物化视图的概念，物化视图是具有物理存储的特殊视图，它占用存储空间，可以进行分区和创建索引等操作。

物化视图是基于表、视图或者其他物化视图创建的。当创建一个物化视图时，Oracle 会自动创建一个内部表来存放物化视图的数据。

8.8.2　什么是查询重写

重写查询，顾名思义就是对 SQL 查询语句进行重写。当用户利用 SQL 语句使用基表进行查询时，如果已经创建了基于这些基表的物化视图，Oracle 将自动计算且使用物化视图来完成查询，毕竟物化视图在某些情况下可以节约查询时间、减少系统 I/O 。我们把 Oracle 的这种查询优化技术称为查询重写。

Oracle 提供基于成本的优化程序（CBO）将自动计算使用物化视图的查询，这些成本包括 CPU 开销、内存使用和系统 I/O 等。如果一个查询涉及多个表的连接操作，则使用物化视图则可以避免连接操作使用的 CPU 和 I/O 开销，减少查询时间。

当然，Oracle 提供了灵活的方式使得用户自己选择是否使用查询重写功能，参数 QUERY_REWRITE_ENABLED 决定是否使用重写查询，该参数为布尔值（Boolean），默认参数值为 false，表示不执行查询重写。在创建物化视图时需要使用 ENABLE QUERY REWRITE 来启

动查询重写功能。下面通过 SHOW 指令查看参数 QUERY_REWRITE_ENABLED 值，如实例 8-25 所示。

【实例 8-25】通过 SHOW 指令查看参数 QUERY_REWRITE_ENABLED 值。

```
SQL> show parameter query_rewrite_enabled

NAME                                TYPE        VALUE
---------------------------------------------------------- ---
query_rewrite_enabled                           string        TRUE
```

参数 QUERY_REWRITE_ENABLED 的值 VALUE 为 TRUE，说明当前运行的数据库允许查询重写功能。

提高查询性能是物化视图的一大优点，Oracle 优化器就是通过代价计算来选择物化视图，通过查询重写来完成用户查询。优化器自动判断一个物化视图是否能满足用户的查询要求，以及是否可以提高查询性能，如果满足要求且可以提高查询性能，优化器就重写用户提交的查询，以使用物化视图，查询重写对用户而言是不可见的。

8.8.3 物化视图的同步

物化视图是基于基表创建的，所以当基表变化时，需要同步数据以更新物化视图中的数据，从而保持物化视图中的数据和基表中的数据的一致性。Oracle 提供了两种物化视图的刷新方式，即 ON COMMIT 方式和 ON DEMAND 方式。

使用 ON COMMIT 方式，当一个基表的变化提交时，则物化视图自动更新，完成与基表的同步。而使用 ON DEMAND 方式时，需要手动同步物化视图和基表数据，此时必须执行 DBMS_MVIEW.REFRESH 过程来同步物化视图。

在选择一种刷新方式后，就可以选择一种刷新类型完成数据同步，刷新类型是指刷新数据时如何实现基表与物化视图的同步，从而将基表的变化反映在物化视图中， Oracle 提供了 4 种刷新类型。

- COMPLETE 类型：该类型将重新执行创建物化视图的 SQL 查询语句，无论基表中修改的数据量是多少，都需要完成一次对物化视图的重新计算。
- FAST 类型：该类型使每个基表的物化视图日志只同步变化了的数据。显然这种方式将节约查询时间成本。
- FORCE 类型：该类型显示使用 FAST 类型更新数据，如果失败，则再使用 COMPLETE 类型刷新整个物化视图。
- NEVER 类型：从不更新，显然这个类型只对那些基表数据不变的物化视图有效。

如果在创建物化视图时，不指定一种刷新类型，则默认使用 FORCE 刷新类型。

图 8-2 给出了使用 FAST 类型刷新数据的物化示意图。

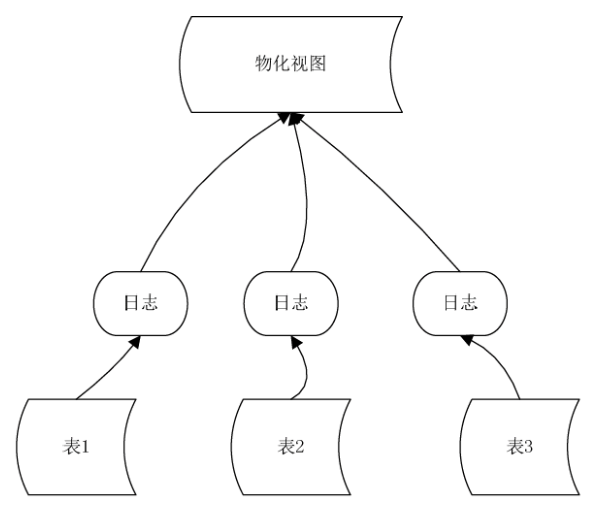


图 8-2　使用 FAST 类型同步物化视图

如图 8-2 所示，一个物化视图涉及到三个表，即表 1、表 2 和表 3，每个表有一个物化视图日志，把自己的变化记录在日志中，而物化视图则通过这些日志文件获得基表的变化，显然这种方式减少了重新执行物化视图的更新时间。



在启动查询重写机制后，Oracle 的查询重写也有可能无法实现，因为无法满足查询重写的某些条件，此时虽然创建了物化视图，但是数据库并不是使用它，显然这样就失去了创建物化视图的作用，一旦这种情况发生，可以使用 DBMS_MVIEW 程序包中的过程进行分析。读者可以参考 Oracle 10g 文档查看该程序包中的过程和使用方法。

8.8.4　物化视图的创建

用户可以根据自己的需要来创建物化视图，当然要求用户必须理解自己的表结构和数据需求。创建物化视图的目的是减少诸如普通视图中由于 JOIN 和 ORDER BY 子句带来的查询耗时问题，以减少系统资源的压力。

1. 创建物化视图的前提条件

创建物化视图的用户必须具有创建物化视图的权限、QUERY REWRITE 的权限以及对创建物化视图所涉及的表的访问权限和创建表的权限。下面，通过 SCOTT 用户来演示上述权限的赋予情况，如实例 8-26 所示。

【实例 8-26】创建物化视图的用户授予权限。

```
SQL> conn system/oracle as sysdba
已连接。
SQL> grant create materialized view to scott;
```

```
授权成功。
SQL> grant query rewrite to scott;
授权成功。
SQL> grant create any table to scott;
授权成功。
SQL> grant select any table to scott;
授权成功。
```

此时，我们使用 DBA 用户为 SCOTT 用户赋予了创建物化视图的所有权限。

2. 创建物化视图日志

物化视图日志是用户选择了刷新类型为 FAST 时要使用的，以同步基表的变化，所以，如果在创建物化视图时选择 FAST 刷新类型，则需要创建该日志，我们计划对 SCOTT 用户的表 DEPT 和表 EMP 创建物化视图，所以要对这两个基表创建物化视图日志，如实例 8-27 所示。

【实例 8-27】针对基表创建物化视图日志。

```
SQL> create materialized view log on dept;
物化视图日志已创建。
SQL> create materialized view log on emp;
物化视图日志已创建。
```

在创建完基表的物化视图日志后，就可以使用创建带有 REFRESH FAST 的参数创建物化视图了。

3. 创建物化视图

创建物化视图的语句很简单，我们通过 CREATE MATERIALIEZDE VIEW 来创建物化视图，这里需要注意各个参数的含义。我们通过实例 8-28 来说明如何创建该视图以及各参数的含义。

【实例 8-28】如何创建物化视图。

```
SQL> create materialized view mtrlview_test
  2  build immediate
  3  refresh fast on commit
  4  enable query rewrite
  5  as
  6  select d.dname,d.loc,e.ename,e.job,e.mgr,e.hiredate,e.sal
  7  from dept d,emp e
  8  where d.deptno = e.deptno;

物化视图已创建。
```

下面详细介绍实例 8-28 中的各参数和子句的作用。

- BUILDE IMMEDIATE：该参数的含义是立即创建物化视图，与立即创建对应的自然是延迟创建的参数，即：BUILD DEFFERED ，该参数说明在物化视图定义后不会立即执行，而是延迟执行，在使用该视图时再创建。
- REFRESH FAST ON COMMIT：FEFRESH FAST 说明刷新数据的类型，选择 FAST 类型，

即快速刷新，该刷新类型要求使用物化视图日志实现与基表数据的同步。ON COMMIT 说明在基表数据提交后立即更新物化视图。

- ENABLE QUERY REWRITE：概述参数的含义是启动查询重写功能，我们已经知道 Oracle 对于查询重写功能的默认选项是不启用的，所以在创建物化视图时明确说明启用查询重写功能。
- AS 子句：该子句定义物化视图的内容，物化视图中该语句查询的结果存储在内部表中。
- 查询体：AS 之后的语句定义了物化视图的查询内容，该 SQL 语句的查询结果输出到物化视图中，保存在由 Oracle 自动创建的表中。

4. 删除物化视图

删除物化视图和删除普通视图相似，不过需要添加一个 MATERIALIZED 关键字，如实例 8-29 所示。

【实例 8-29】删除物化视图。

```
SQL> drop materialized view mtrlview_test;
物化视图已删除。
```

8.8.5　物化视图的使用环境

从以上对物化视图的定义、分析，以及创建和管理物化视图的实例中，我们已经体会到物化视图的基本作用，可以这样总结：物化视图是一种可以用于汇总、计算、复制以及发布数据的方案。与以上行为对应，物化视图适用于数据仓库、分布式计算以及移动计算等环境。下面依次讨论这 3 个应用环境。

1. 数据仓库

在数据仓库中往往需要存储对于基表的汇总或平均数据等，物化视图用户进行类似的计算来存储聚合后的数据。因为在数据仓库中的物化视图内通常存储汇总数据，所以数据仓库中的物化视图也称为概要。

2. 分布式环境

在分布式环境中可以通过物化视图实现不同节点间的数据同步，使得同样的数据分布在不同的物理空间，更好地响应用户的查询，减少中心数据库服务器的负担。物化视图的复制功能使得用户可以把数据拷贝到远程节点，而数据同步能力又可以同步各个节点间的数据，从而实现了数据的本地访问。为了形象地说明物化视图在分布式环境中的应用，给出如图 8-3 所示的示意图。

在图 8-3 中无论用户 1 还是用户 2，他们查询中心服务器的数据时，都是通过查询本地的物化视图实现的，物化视图通过数据同步机制更新数据，对于用户而言这个过程是透明的，用户认为是通过远程的中心服务器实现了该查询。

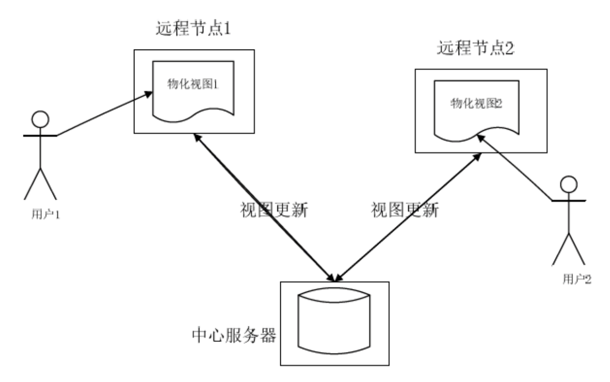


图 8-3　物化视图在分布式环境中的应用

3. 移动计算

该环境中也是利用物化视图的优化查询功能来节约查询时间，同时移动客户端使用物化视图下载一个数据子集，也可以定期地从中央服务器获得新数据，在客户端修改后发到中央服务器。

显然在生产数据库中物化视图是非常重要的应用，尤其在分布式环境和移动计算环境中，物化视图的分布式存储和同步功能很好地满足了客户对于本地读取数据的需求，更好地响应客户对于数据请求的时间要求。

8.9　本章小结

视图极大地方便了用户对于数据的操作，不但增加了对表访问的安全性，而且减少了很多复杂的查询过程，在本章前半部分重点讲了视图的概念、如何创建和修改视图。Oracle 的视图管理部分使读者可以把握通过数据字典查看视图的方式，以及视图执行的内部过程。对于在普通视图上的 DML 操作，Oracle 做了限制，对于简单视图可以使用 DML 操作，而对于复杂视图则必须遵守一定的原则或条件。

在本章的后半部分我们引入了物化视图，物化视图是 Oracle 10g 中提出的新的概念，需要读者理解物化视图的本质，物化视图具有存储空间，在一定条件下可以实现 DML 操作，使用同步机制实现与创建物化视图的表的数据同步，物化视图在数据仓库、分布式计算和移动计算中都具有很重要的作用。

第 9 章

◀ 事　　务 ▶

事务是数据库领域中一个非常重要的概念，早期的数据库都是联机事务处理系统，即：OLTP，所以对事务都有很好地支持，事务是数据库处理的核心，DBMS 使用事务协调用户的并发行为，减少用户访问资源的冲突。

9.1　什么是事务

在各种数据库教材中都使用银行取款的实例来说明事务的作用，笔者也引用大家习以为常的行为来分析事务的概念和作用。

如果用户 A 要给用户 B 从银行转账 10000 元，此时我们考虑 ATM 机的行为，把 ATM 机的行为作为一个事务。ATM 机的实施步骤如下：

01 从用户 A 的账户减少 1000 元。

02 向用户 B 的账户增加 1000 元。

上述两个步骤必须都成功执行，如果两个步骤任何一个出现问题，ATM 机都没有正确完成这次转账行为。谁也不希望在自己的账户上白白丢失 10000 元吧。此时 ATM 机的两个执行步骤是不可分割的，它要么执行成功，要么不执行（回滚所有更改的数据），ATM 机的两个操作在逻辑上就可以看做是一个完整的事务。

本节给出事务的一个更加详细的说明：事务是一组逻辑工作单元，它由一条或多条 SQL 语句组成。一个事务可以在操作的数据库对象上执行一个或多个操作，事务可以作为程序的部分功能而执行。事务开始于一条可执行的 SQL 语句，继续执行事务主体，然后结束于以下的一种情况发生。

- 显式提交 COMMIT：当事务遇到 COMMIT 指令时，将结束事务并永久保存所有更改的数据到数据库文件中。
- 显式回滚 ROLLBACK：当事务遇到 ROLLBACK 指令时，也结束事务的执行，但是此时它回滚所有更改的数据到其原始值，即取消所有更改。

- DDL 语句：一旦用户在使用数据定义语言时，如 CREATE、DROP 等，则之前的所有 DML 语言操作都作为事务的一部分而提交，此时称为隐式提交。
- 正常结束程序：如果 Oracle 数据库应用程序正常结束，如使用 SQL*Plus 工具更改了数据，而正常退出该工具程序，则 Oracle 自动提交事务。
- 非正常地结束程序：当程序崩溃或意外中止时，所有数据更改都被回滚，类似于显式回滚操作的结果，这里是隐式回滚的，因为没有用户参与。

事务的 4 个特性，简写为 ACID 特性，即 4 个特性的英文首字母。下面详细介绍事务的 4 个特性并说明 Oracle 是如何实现的。

- 原子性（Atomicity）：事务要么执行成功，要么什么也不执行。如果事务执行了一部分而系统崩溃或发生异常，则 Oracle 将回滚所有更改的数据，此时 Oracle 使用还原段管理更改数据的原始值用于事务回滚。
- 一致性（Consistency）：事务必须将数据库保持在一致状态，如在 SCOTT 用户的 DEPT 表中删除一条记录，但是 EMP 表中存在雇员属于要删除的部门，那就拒绝这样的操作执行，即数据库中的数据保持在一致状态。
- 隔离性（Isolation）：隔离性使得多个用户隔离执行实现数据库的并发访问。这种隔离性要求一个事务修改的数据在未提交前，其他事务看不到它所做的更改。Oracle 使用并发控制机制实现事务的隔离性。
- 持久性（Durability）：该特性保证提交的事务永久地保存在数据库中，在 Oracle 数据库中提交的数据并不是立即写入数据文件，而是先保存在数据库高速缓存中，为了防止实例崩溃，Oracle 使用日志优先的方法，首先将提交的数据更改写入重做日志文件，即使实例崩溃也可以在实例恢复时保证事务的持久性。

9.2 事务控制

事务控制使得用户可以控制事务的行为，事务控制有显式和隐式之分，显式控制是指用户使用显式控制指令控制事务，隐式控制是指在某些特定条件下，如 DDL 语句发生时、程序正常退出时对事务的行为进行控制。

9.2.1 使用 COMMIT 实现事务控制

当用户修改数据时，如果想显式提交更改结果，此时可以使用 COMMIT 语句提交更改，在用户更改数据时，如果没有提交数据则其他用户看不到该事务所做的数据更改，只有用户提交了更改（无论显式还是隐式的），其他用户才可以看到数据变化，如实例 9-1 所示。

【实例 9-1】查看表 DEPT 的内容。

```
SQL> select *
  2  from dept;
```

```
   DEPTNO DNAME          LOC
------------------ ------------------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH        DALLAS
       30 SALES           CHICAGO
       40 OPERATIONS      BOSTON
```

从输出可以看到表 DEPT 有 4 条记录，下面我们执行一个事务，该事务的作用是从表 DEPT 中删除 DEPTNO 为 40 的记录。

【实例 9-2】删除表 DEPT 中的一条记录。

```
SQL> delete from dept
  2  where deptno = 40;

已删除 1 行。
```

此时，提示已经删除了 DEPTNO=40 的记录，但是此时无论显式和隐式我们都没有提交数据更改，所以其他用户不应该看到更改后的数据，即其他用户仍然会看到 DEPTNO=40 的记录。

【实例 9-3】用 SYSTEM 用户登录数据库并查看表 DEPT 的表数据。

```
SQL> conn system/oracle@orcl
已连接。
SQL> select *
  2  from scott.dept;

   DEPTNO DNAME          LOC
----------------- ------------------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
```

正如预料的，SYSTEM 用户可以看到 DEPTNO=40 的记录，这也是 Oracle 实现事务隔离性的体现。下面在实例 9-2 执行步骤之后显式提交更改，如实例 9-4 所示。

【实例 9-4】显式提交实例 9-2 的更改。

```
SQL> commit;

提交完成。
```

此时提交完成，注意此时提交后，更改的数据可能并没有写入数据文件，但是由于重做日志的保护，不影响事务的持久性。在检验其他用户是否可以看到提交后的数据，即用户应该无法看到 DEPTNO=40 的记录，如实例 9-5 所示。

【实例 9-5】利用 SYSTEM 用户登录数据库并查看实例 9-2 提交后的结果。

```
SQL> conn system/oracle@orcl
已连接。
```

```
SQL> select *
  2  from scott.dept;

    DEPTNO DNAME          LOC
----------------- ----------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
```

在用户提交了数据更改后，我们通过 SYSTEM 用户登录数据库，此时可以看到由于事务提交了数据更改，所以其他用户看到的是更改后的数据，即 DEPTNO=40 的记录成功删除了。

用户通过 COMMIT 来提交一个事务，完成事务的持久性，那么当显式地提交前后 Oracle 到底做了哪些工作呢？

在用户显式提交前，Oracle 数据库内部发生了如下行为：

- 在非系统还原段中生成要更改的数据的备份。
- 在重做日志缓冲区中创建重做日志选项。
- 在数据库高速缓冲区修改数据（删除或更新）。

在用户显式提交后，Oracle 数据库内部发生如下行为：

- 在重做记录的事务表中标记上已提交事务的 SCN，说明该事务已经提交了。
- LGWR 将事务的重做日志信息和已提交事务的 SCN 号写入重做日志文件。此时，认为提交完成了。
- 释放 Oracle 持有的对更改的数据对象的锁，标记事务完成。

9.2.2 使用 ROLLBACK 实现事务控制

ROLLBACK 回滚所有没有提交的数据更改，Oracle 使用还原段实现回滚功能。由于用户直接使用 ROLLBACK 回滚数据，所以称此为显式的事务控制——事务回滚。我们还是通过实例来说明。下面的操作都是在 SCOTT 用户模式下实现的。我们向表 DEPT 中插入一行记录，如实例 9-6 所示。

【实例 9-6】向表 DEPT 中插入一行记录。

```
SQL> insert into scott.dept
  2  values(40,'OPERATIONS','BOSTON');

已创建 1 行。
```

此时显示已经成功插入一行记录，其实就是我们在实例 9-2 中删除的那行记录。下面的查询直接在实例 9-6 执行后运行。

【实例 9-7】查询插入记录后表 DEPT 中的数据。

```
SQL> select *
  2  from scott.dept;
```

```
   DEPTNO DNAME          LOC
------------------ ------------------------- -------------
       40 OPERATIONS      BOSTON
       10 ACCOUNTING     NEW YORK
       20 RESEARCH        DALLAS
       30 SALES            CHICAGO
```

我们看到，查询结果说明已经向表中插入了数据，该行记录显示在第一行。如果此时用户要撤销刚才的插入操作，则可以直接输入 ROLLBACK 指令结束刚才插入记录的事务。

【实例 9-8】回滚实例 9-6 中的插入数据记录事务。

```
SQL> rollback;

回退已完成。
```

提示回退完成，此时 Oracle 使用重做表空间中的重做记录来恢复数据。我们查看一下回滚事务的结果，如实例 9-9 所示。

【实例 9-9】查询表 DEPT 在事务回滚后的数据。

```
SQL> select *
  2  from scott.dept;

   DEPTNO DNAME          LOC
------------------ ------------------------ -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES            CHICAGO
```

从输出可以看出，没发现 DEPTNO =40 的那行记录，说明事务回滚成功。

9.2.3 程序异常退出对事务的影响

在事务执行过程中，程序会发生异常，如实例崩溃等，此时事务结束并回滚所有的数据更改。我们使用 SQL*Plus 的工具登录数据库并执行一个事务，然后不是正常退出而是关闭 DOS 对话框模拟程序异常，查看事务如何处理，如实例 9-10 所示打开 SQL*Plus 并登录数据库。

【实例 9-10】打开 SQL*Plus 并登录数据库到 SCOTT 模式。

```
C:\Documents and Settings\Administrator>sqlplus /nolog

QL*Plus: Release 11.1.0.6.0 - Production on 星期一 8月 10 16:41:18 2009

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> conn scott/oracle@orcl
已连接。
SQL>
```

此时，我们执行一个事务，在表 DEPT 中增加一条记录，如实例 9-11 所示。

【实例 9-11】向表 DEPT 添加一条记录。

```
SQL> insert into dept
  2  values(50,'MARKETING','BOSTON');

已创建 1 行。
```

此时成功向表 DEPT 中添加一条记录，然后不提交更改，而是关闭 DOS 窗口，即关闭如图 9-1 所示的对话框。

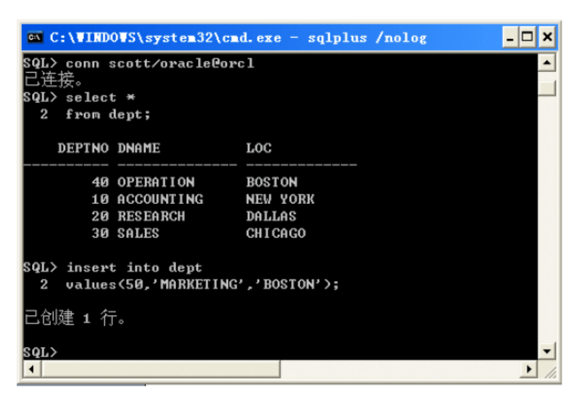


图 9-1 向表 DEPT 中插入一行记录

然后重新登录数据库到 SCOTT 模式，再查询表 DEPT 中的数据，看是否成功添加了一条记录，该记录的 DEPTNO = 50，如实例 9-12 所示。

【实例 9-12】查询程序异常退出后插入操作的事务是否成功。

```
SQL> select *
  2  from dept;

   DEPTNO  DNAME          LOC
----------------- -------------------------- -------------
        40  OPERATION      BOSTON
        10  ACCOUNTING     NEW YORK
        20  RESEARCH       DALLAS
        30  SALES          CHICAGO
```

我们再重新向表 DEPT 中插入一条记录，和图 9-1 中插入的记录一样，不过这次，我们使用指令 EXIT 正常退出 SQL*Plus，然后再登录数据库查看是否成功插入数据，即事务是否隐式提交。

【实例 9-13】向表 DEPT 插入一条数据并正常退出程序。

```
SQL> insert into dept
  2  values (50,'MARKETING','BOSTON');

已创建 1 行。
```

```
SQL> exit
从 Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options 断开

C:\Documents and Settings\Administrator>
```

然后，使用 SCOTT 用户登录数据库，查询表 DEPT 的内容确认事务是否成功提交。

【实例 9-14】程序正常退出后查询事务是否成功执行。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> select *
  2  from dept;

    DEPTNO DNAME          LOC
------------------ -------------------------- -------------
        40 OPERATION       BOSTON
        50 MARKETING      BOSTON
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
```

此时，表 DEPT 的记录中多了第二行记录，即 DEPTNO=50 的记录，该记录是我们在实例 9-13 中插入的，因为正常退出程序，所以 Oracle 隐式提交了数据更改。

9.2.4 使用 AUTOCOMMIT 实现事务自动提交

Oracle 提供了一种自动提交 DML 操作的方式，这样一旦用户执行了 DML 操作，如 UPDATE、DELETE 等，数据就自动提交。

【实例 9-15】设置数据库服务器的 AUTOCOMMIT 模式。

```
SQL> conn system/oracle@orcl
已连接。
SQL> set autocommit on;
```

在执行 set autocommit on 指令后，不会有任何提示，不过当执行如 DELETE 操作时，数据更改会自动提交。

【实例 9-16】在自动提交模式下执行事务。

```
SQL> delete from scott.dept
  2  where deptno=50;

已删除 1 行。

提交完成。
```

删除 SCOTT 用户的表 DEPT 中 DEPTNO = 50 的记录时，没有使用显式提交，也没有任何隐式提交的事件发生，此时处于自动提交模式，所以事务自动提交，这与实例 9-2 不同。

如果不需要自动提交，可以关闭自动提交，Oracle 默认是非自动提交。

【实例 9-17】关闭自动提交。

```
SQL> set autocommit off;
SQL> delete from scott.dept
  2  where deptno = 40;

已删除 1 行。
```

此时，没有提示“提交完成”，说明成功关闭了自动提交。为了方便以后的数据操作，我们使用 ROLLBACK 回滚事务。

【实例 9-18】回滚事务。

```
SQL> rollback;

回退已完成。
```

再查询表 DEPT 的数据，查看 DEPTNO= 40 的记录是否存在。

【实例 9-19】查询事务回滚后表 DETP 中的数据。

```
SQL> select *
  2  from scott.dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATION      BOSTON
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
```

我们看到表 DEPT 中原来删除的记录（DEPTNO=40）由于事务回滚，依然存在。

9.3 本章小结

本章只讲了事务的概念，事务是一个逻辑工作单元，由一条或多条 SQL 语句组成，事务作用于 Oracle 对象上，如查询表、建立表空间等，由于早期的数据库都是 OLTP 系统，所以对事务有很好地支持。事务的 4 个特性（ACID），即：原子性、一致性、隔离性和持久性需要读者认真理解体会。

事务控制使得用户可以更加自由地控制事务的行为，Oracle 模式处于非自动提交模式时，即用户事务中执行了 DML 操作后并不是提交该数据，而是需要用户显式或隐式的提交数据更改。在没有提交数据更改时，ROLLBACK 指令可以使得事务回滚，另外在 Oracle 程序非正常退出或实例崩溃时，事务会自动回滚，如果程序正常退出时，事务隐式提交。

第 10 章

数据查询

数据查询是数据库中最基本、最常用的一类数据操作，数据查询可以由用户也可以由应用程序来完成。本章将讲解 Oracle 的数据查询语法格式、简单的数据查询和复杂的数据查询，复杂查询包括连接查询、子查询和联合查询。并在每节都给出恰当的实例和直观地说明，使得读者可以轻松地掌握数据查询的原理和查询功能的实现。

10.1 简单查询

简单查询只涉及到单个表的查询，本节为了引导读者入门，先介绍简单查询，重点包括如何实现简单查询和查询规范、使用别名机制和使用连接运算符、如何使用 DISTICT 语句以及在简单查询中使用算数运算。下面将依次介绍这些内容。

1. 实现简单查询及规范

简单查询只涉及一个表，包括两个关键字 SELECT 和 FROM。其基本语法格式为：

```
SELECT 列名[列名……] FROM 表名
```

下面给出一个实例来说明如何实现简单查询。使用 SCOTT 用户登录数据库服务器，查询 EMP 表中的员工信息。在查询一个表之前，往往需要知道该表中到底有哪些列属性，此时读者使用 describe 指令查看表的列属性信息，作为一章的开始，我们给出完整的登录和查询过程，如实例 10-1 所示。

【实例 10-1】登录数据库并查询用户 SCOTT 的表 EMP 的结构。

```
C:\Documents and Settings\Administrator>sqlplus /nolog
QL*Plus: Release 11.1.0.6.0 - Production on 星期五 7月 24 08:35:31
Copyright (c) 1982, 2007, Oracle.  All rights reserved.
SQL> conn scott/oracle
已连接。
SQL> desc emp;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ---------------
```

```
EMPNO                                    NOT NULL NUMBER(4)
ENAME                                             VARCHAR2(10)
JOB                                               VARCHAR2(9)
MGR                                               NUMBER(4)
HIREDATE                                          DATE
SAL                                               NUMBER(7,2)
COMM                                              NUMBER(7,2)
DEPTNO                                            NUMBER(2)
```

我们看到表 EMP 有 8 个列属性，DESC EMP 指令输出列名，通过实例 10-2 查询表 EMP 中的员工信息。

【实例 10-2】查询表 EMP 中的员工信息。

```
SQL> SELECT empno,ename,job,sal,deptno
  2  FROM emp;

    EMPNO ENAME      JOB              SAL     DEPTNO
------------------ ------------------- ------------------------ ----------
     7369 SMITH      CLERK            800         20
     7499 ALLEN      SALESMAN        1600         30
     7521 WARD       SALESMAN        1250         30
     7566 JONES      MANAGER         2975         20
     7654 MARTIN     SALESMAN        1250         30
     7698 BLAKE      MANAGER         2850         30
     7782 CLARK      MANAGER         2450         10
     7788 SCOTT      ANALYST         3000         20
     7839 KING       PRESIDENT       5000         10
     7844 TURNER     SALESMAN        1500         30

    EMPNO ENAME      JOB              SAL     DEPTNO
------------------ ------------------- ------------------------ ----------
     7876 ADAMS      CLERK           1100         20
     7900 JAMES      CLERK            950         30
     7902 FORD       ANALYST         3000         20
     7934 MILLER     CLERK           1300         10

已选择 14 行。
```

这里对书写 SQL 语句做以下说明：

- 最好是关键字大写，而列名小写，这样便于阅读。
- SQL 的关键字不能缩写，必须严格按照原关键字书写。
- SQL 语句可以单行排列，也可以多行排列，但是为了便于阅读推荐多行排列。
- 列名的大小写不敏感，而且可以在同一个列名中大小写混排，不影响数据查询，如把 ename 改为 EnaME。

2. 别名及连接符

在实例 10-2 的输出中，我们看到的表名很不好理解，至少不是很直观，对于一个不懂数据

库的人而言，看到这样的报表输出显然难于理解，如果可以将表名修改成中文或更易于理解的英文就好了。

（1）使用别名

Oracle 提供了一个别名机制来实现列名的修改，如实例 10-3 所示。

【实例 10-3】使用别名机制来修改列名。

```
SQL> SELECT empno "员工号",ename "姓名", job "岗位", sal "月工资", deptno
"部门号"
  2  FROM emp;

    员工号 姓名                岗位             月工资      部门号
------------------ ------------------- ------------------------ ----------
      7369 SMITH       CLERK           800         20
      7499 ALLEN      SALESMAN        1600         30
      7521 WARD       SALESMAN        1250         30
      7566 JONES       MANAGER         2975         20
      7654 MARTIN      SALESMAN        1250         30
      7698 BLAKE      MANAGER         2850         30
      7782 CLARK      MANAGER         2450         10
      7788 SCOTT      ANALYST          3000         20
      7839 KING        PRESIDENT        5000         10
      7844 TURNER      SALESMAN        1500         30

    员工号 姓名                岗位             月工资      部门号
------------------ ------------------- ------------------------ ----------
      7876 ADAMS      CLERK           1100         20
      7900 JAMES       CLERK            950         30
      7902 FORD        ANALYST        3000         20
      7934 MILLER      CLERK            1300         10

已选择 14 行。
```

设置别名的方法是在列名后空格，然后使用双引号，在双引号内输入别名，如果是英文别名可以在列名后空格，然后紧跟着英文别名，如实例 10-4 所示。

【实例 10-4】使用别名查询数据。

```
SQL> select empno employee_number,ename employee_name,sal salary
  2  from emp;

EMPLOYEE_NUMBER EMPLOYEE_N     SALARY
------------------------------- ------------------ --------------------
              7369 SMITH             800
              7499 ALLEN            1600
              7521 WARD             1250
              7566 JONES             2975
…….
```

（2）使用连接符

连接符顾名思义是起到连接作用的，在查询输出中为了使得输出结果更易于阅读，更像自然语言的方式，使用连接符可达到这个效果，如实例 10-5 所示。

【实例 10-5】使用连接运算符。

```
SQL> SELECT ename ||'的雇佣日期是: ' || hiredate
  2  FROM emp;

ENAME||'的雇佣日期是: '||HIREDATE
----------------------------------------
SMITH 的雇佣日期是: 17-12 月-80
ALLEN 的雇佣日期是: 20-2 月 -81
WARD 的雇佣日期是: 22-2 月 -81
JONES 的雇佣日期是: 02-4 月 -81
MARTIN 的雇佣日期是: 28-9 月 -81
BLAKE 的雇佣日期是: 01-5 月 -81
CLARK 的雇佣日期是: 09-6 月 -81
SCOTT 的雇佣日期是: 19-4 月 -87
KING 的雇佣日期是: 17-11 月-81
TURNER 的雇佣日期是: 08-9 月 -81

ENAME||'的雇佣日期是: '||HIREDATE
----------------------------------------
ADAMS 的雇佣日期是: 23-5 月 -87
JAMES 的雇佣日期是: 03-12 月-81
FORD 的雇佣日期是: 03-12 月-81
MILLER 的雇佣日期是: 23-1 月 -82

已选择 14 行。
```

使用连接符“||”把多个列和字符串连接起来，连接的字符串是字符型或日期型，则必须用单引号括起来。其实在使用连接符的查询语句中，也可以使用别名更好地表达查询意图，如实例 10-6 所示，我们重新查询实例 10-5 的信息，不过此时使用了别名。

【实例 10-6】使用别名和连接符号“||”实现查询。

```
SQL> SELECT ename ||'的雇佣日期是: ' || hiredate "员工雇佣日期查询"
  2  FROM emp;

员工雇佣日期查询
----------------------------------------
SMITH 的雇佣日期是: 17-12 月-80
ALLEN 的雇佣日期是: 20-2 月 -81
……
```

上述输出结果中省略了部分输出以节约篇幅。我们看到此时输出的列名已经变为设置的别名“员工雇佣日期查询”，这样的别名更易于理解和阅读。

3. 使用 DISTINCT 语句

DICTINCT 的运算符使得输出的结果中对于某列的值不允许重复，也就是该列的输出值应该是独一无二的。为了说明它的用法，下面给出实例。

先观察实例 10-7 的 SQL 语句和输出结果。

【实例 10-7】查询表 EMP 的 JOB 列数值。

```
SQL> select job
  2  from emp;

JOB
---------
CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
ANALYST
PRESIDENT
SALESMAN

JOB
---------
CLERK
CLERK
ANALYST
CLERK

已选择 14 行。
```

该实例中，我们查询表 EMP 中 JOB 的列值发现有 14 个，但是很多是重复的值，如果想在表 EMP 中计算有多少个 JOB，则不需要重复的值。可以利用 DISCTINCT 来实现这个想法，如实例 10-8 所示。

【实例 10-8】使用 DISTINCT 查询 JOB 列值。

```
SQL> SELECT DISTINCT job
  2  FROM emp;

JOB
---------
CLERK
SALESMAN
PRESIDENT
MANAGER
ANALYST
```

显然此时的输出没有重复的值，显示当前表 EMP 中有 4 个 JOB。为了计算并输出不同 JOB

的数量，我们使用函数 COUNT 结合 DISTINCT 运算符，如实例 10-9 所示。

【实例 10-9】用函数 COUNT 结合 DISTINCT 运算符查询不同类型的 JOB 数。

```
SQL>  SELECT COUNT(DISTINCT job) "不同的岗位数量"
  2  FROM emp;

不同的岗位数量
---------------------
            5
```

4. 简单查询中的算数运算

在查询中自然可以使用算数运算符，即加、减、乘、除四则运算。我们继续对表 EMP 进行操作，如对所有员工的工资统一增加 10%，如实例 10-10 所示。

【实例 10-10】在查询中使用运算符。

```
SQL>  SELECT ename ||'增加 10%后的月薪是：'|| sal*(1+0.1) "加薪后的员工工资"
  2  FROM emp;

加薪后的员工工资
----------------------------------------------------------------------
SMITH 增加 10%后的月薪是：880
ALLEN 增加 10%后的月薪是：1760
WARD 增加 10%后的月薪是：1375
JONES 增加 10%后的月薪是：3272.5
MARTIN 增加 10%后的月薪是：1375
BLAKE 增加 10%后的月薪是：3135
CLARK 增加 10%后的月薪是：2695
SCOTT 增加 10%后的月薪是：3300
KING 增加 10%后的月薪是：5500
TURNER 增加 10%后的月薪是：1650

加薪后的员工工资
----------------------------------------------------------------------
ADAMS 增加 10%后的月薪是：1210
JAMES 增加 10%后的月薪是：1045
FORD 增加 10%后的月薪是：3300
MILLER 增加 10%后的月薪是：1430

已选择 14 行。
```

在这个实例中，我们综合使用了连接符、运算符和别名机制实现了简单查询，将表 EMP 中员工的工资统一增加 10%，然后输出一个易读的计算结果。

如果读者学习过某种计算机语言，如 C、C++或 Java 等，此处的运算规则和它们一样，即先乘除后加减；同级运算符按照从左到右的顺序计算；如果有括号，括号中的计算优先等。

10.2　条件查询

在简单查询中，我们查询了表中相关列的所有记录，但是生产数据库中往往需要查询满足一定条件的记录，如查询岗位 JOB 为 MANAGER 的员工信息等，此时需要使用条件查询，条件查询使用关键字 WHERE 实现。条件查询的语句格式为：

```
SELECT 列名 [列名……]
FROM 表名
WHERE 条件语句
```

【实例 10-11】查询岗位 JOB 为 MANAGER 的员工信息。

```
SQL> SELECT job,ename,sal,hiredate
  2  FROM emp
  3  WHERE job = 'MANAGER';

JOB        ENAME              SAL  HIREDATE
---------------- -------------------- -------------- ----------------------
MANAGER    TOM               2000  21-7月 -09
MANAGER    JONES             2975  02-4月 -81
MANAGER    BLAKE             2850  01-5月 -81
MANAGER    CLARK             2450  09-6月 -81
```

WHERE 子句限制了查询的输出。在条件查询中 WHERE 后可以跟各种条件，如比较关系运算符，比较关系运算符包括：

- >（大于）。
- <（小于）。
- >=（大于等于）。
- <=（小于等于）。
- =（等于）。
- =!或<>（不等于）。

在 WHERE 子句中，只要是合法的语句或函数都可以使用。我们给出一个使用 IN 操作符的条件查询实例，如实例 10-12 所示。

【实例 10-12】使用 IN 操作符的条件查询。

```
SQL>SELECT ename,job,hiredate,sal
  2  FROM emp
  3  WHERE job in ('MANAGER','SALESMAN')
  4  ORDER BY sal;

ENAME      JOB       HIREDATE              SAL
---------- --------- -------------- ----------
WARD       SALESMAN  22-2月 -81           1250
MARTIN     SALESMAN  28-9月 -81           1250
```

```
TURNER     SALESMAN  08-9月 -81          1500
ALLEN      SALESMAN  20-2月 -81          1600
TOM        MANAGER   21-7月 -09          2000
CLARK      MANAGER   09-6月 -81          2450
BLAKE      MANAGER   01-5月 -81          2850
JONES      MANAGER   02-4月 -81          2975

已选择 8 行。
```

该实例的目的是查询表 EMP 中 JOB 为 MANAGER 或 SALESMAN 的员工信息，并且按照工资增序排列。它的查询过程是：首先 Oracle 先选择满足条件的行记录，然后按照选择的列进行投影操作，选择出符合条件的员工的属性信息，然后 ORDER BY 子句按照工资增序进行查询结果的排序，最后输出给用户。

下面我们总结一下在 WHERE 子句中类似于 IN 的比较运算符。

- IN（包含任何一个，即满足条件）。
- =ANY（相当于 IN）。
- >ANY（大于最小的）。
- <ANY（小于最大的）。
- >ALL（大于最大的）。
- <ALL（小于最小的）。

我们再给出使用 ANY 和 ALL 的实例，以方便读者理解，如实例 10-13 所示。

【实例 10-13】使用 ANY 和 ALL 实现查询。

```
SQL> select ename,job,hiredate,sal
  2  from emp
  3* where sal >any (2000,3000)

ENAME      JOB       HIREDATE            SAL
-------------------- --------------- ------------------------------
JONES      MANAGER   02-4月 -81          2975
BLAKE      MANAGER   01-5月 -81          2850
CLARK      MANAGER   09-6月 -81          2450
SCOTT      ANALYST   19-4月 -87          3000
KING       PRESIDENT  17-11月-81          5000
FORD       ANALYST   03-12月-81          3000

已选择 6 行。
```

实例 10-13 输出工资多于 2000 的员工信息，相当于 WHERE SAL >2000。我们把 ANY 换成 ALL，此时查询工资大于 3000 的员工信息，如实例 10-14 所示。

【实例 10-14】使用 ALL 查询工资大于 3000 的员工信息。

```
SQL> select ename,job,hiredate,sal
  2  from emp
  3* where sal >all (2000,3000)
```

```
ENAME      JOB        HIREDATE            SAL
------------------- --------------- -----
KING       PRESIDENT   17-11 月-81          5000
```

该实例类似于 WHERE SAL>3000，此时我们看不到使用 ANY 和 ALL 的好处，问题是我们在 ALL 和 ANY 中都明确给出了数值，但是在实际中并不是都可以给出明确的数据，而是一个未知的子查询，这样使用 ALL 和 ANY 就体现出优越性了，下节我们将着重讲解子查询。

10.3　子查询

子查询是在查询中包含查询的一种数据查询方式，子查询分为单行子查询和多行子查询，这里单行和多行的意思是子查询中返回结果涉及的行数，子查询可以放在 WHERE 子句后，而对于单行子查询还可以放在 HAVING 子句和 FROM 子句中。

1. 单行子查询

（1）WHERE

单行子查询是指在 WHERE 子查询中的查询结果是单行数据，如 SELECT job FROM emp WHERE ename ='SCOTT'就是一个单行子查询，因为它只返回一行数据。下面通过另一个实例演示如何使用子查询。我们查找和员工 SCOTT 具有相同工作岗位的员工信息，此时的岗位选择使用子查询，如实例 10-15 所示。

【实例 10-15】在 WHERE 子句中使用子查询。

```
SQL> select ename,job,hiredate,sal
  2  from emp
  3  where job = (select job
  4          from emp
  5              where ename ='SCOTT');

ENAME      JOB       HIREDATE            SAL
----------------- ----------------- ------------------------
SCOTT      ANALYST    19-4 月 -87          3000
FORD       ANALYST   03-12 月-81           3000
```

在实例 10-15 中，WHERE 子句中的查询称为子查询，而外面的查询称为主查询，主查询的查询过程是：首先执行子查询，找到 JOB 列的值，该值只有一个查询结果，即 ANALYST，然后执行主查询。

在 WHERE 子查询中，也可以使用多个单行子查询组成更具体的查询条件，如实例 10-16 所示使用 AND 运算。

【实例 10-16】在子查询中使用条件和 AND 运算。

```
SQL> conn scott/oracle@orcl
```

```
已连接。
SQL> select ename,job,hiredate,sal
  2  from emp
  3  where job = (select job
  4               from emp
  5               where ename ='SMITH')
  6  and sal <=(select avg(sal)
  7           from emp);

ENAME      JOB       HIREDATE              SAL
---------- --------- -------------- ----------
SMITH      CLERK     17-12 月-80           800
ADAMS      CLERK     23-5 月 -87          1100
JAMES      CLERK     03-12 月-81           950
MILLER     CLERK     23-1 月 -82          1300
```

该查询的两个子查询中，一个是查询和员工 SMITH 相同的 JOB，另一个是查询 EMP 表中员工的平均工资。主查询的目的是查询 JOB 与员工 SMITH 相同且工资小于员工平均工资的员工信息。

（2）HAVING

HAVING 子句用来限制分组函数，我们先给出一个使用 HAVING 子句限制分组函数的实例，然后再给出在 HAVING 子句中使用单行子查询的实例。

【实例 10-17】使用 HAVING 子句限制分组函数。

```
SSQL> select job,min(sal),avg(sal),max(sal)
  2  from emp
  3  group by job
  4  having avg(sal)>2000;

JOB         MIN(SAL)   AVG(SAL)   MAX(SAL)
--------- ---------- ---------- ----------
PRESIDENT       5000       5000       5000
MANAGER         2450 2758.33333       2975
ANALYST         3000       3000       3000
```

上述查询中，HAVING 子句限制了分组函数 AVG(sal)的输出条件，只有满足 AVG(sal)>2000 时，才输出结果，目的是查询表 EMP 中平均工资大于 2000，且按照工作岗位 JOB 分类后的每类岗位的最低工资、平均工资和最高工资。

此时我们可以在 HAVING 子句中使用单行子查询代替实例 10-17 中 HAVING 子句中的条件，如实例 10-18 所示。

【实例 10-18】使用单行子查询代替实例 10-17 中 HAVING 子句中的条件查询。

```
SQL> select job,min(sal),avg(sal),max(sal)
  2  from emp
  3  group by job
  4  having avg(sal) >(
  5                  select min(avg(sal))
  6                  from emp
```

```
  7                   group by job);

JOB        MIN(SAL)   AVG(SAL)   MAX(SAL)
----------------- ------------------- ----------------- ------------------
SALESMAN        1250       1400       1600
PRESIDENT        5000       5000       5000
MANAGER         2450  2758.33333       2975
ANALYST         3000       3000       3000
```

该查询中，HAVING 子句使用了一个子查询，查询表 EMP 中按照岗位 JOB 分类的最少的平均工资。

其实，我们可以分开查询，先查询表 EMP 中按照岗位 JOB 分类的最少的平均工资，再根据这个最少的平均工资执行 HAVING 子句的条件查询，如实例 10-19 所示。

【实例 10-19】查询表 EMP 中按照岗位 JOB 分类的最少的平均工资。

```
SQL> select min(avg(sal))
  2  from emp
  3  group by job;

MIN(AVG(SAL))
--------------------------
      1037.5

SQL> select job,min(sal),avg(sal),max(sal)
  2  from emp
  3  group by job
  4  having avg(sal) >1037.5;

JOB        MIN(SAL)   AVG(SAL)   MAX(SAL)
--------- ---------- ---------- ----------
SALESMAN        1250       1400       1600
PRESIDENT       5000       5000       5000
MANAGER         2450 2758.33333       2975
ANALYST         3000       3000       3000
```

其实，实例 10-19 和实例 10-18 查询的结果一样，使用 HAVING 子句的子查询使得在一个 SQL 语句中执行了更复杂的条件查询，子查询也为条件数值不确定的查询提供了很好地解决方案。

（3）FROM

FROM 子句后跟的是表名，在一定条件下也可以使用单行子查询，例如查询表 EMP 中工资大于平均工资的所有员工的信息，如实例 10-20 所示。

【实例 10-20】查询表 EMP 中工资大于平均工资的所有员工的信息。

```
SQL> select e.ename,e.sal,e.job,d.av sal
  2  from emp e,(select job,avg(sal) av_sal
  3            from emp
  4            group by job) d
  5  where e.job = d.job
```

```
  6* and e.sal >d.av_sal

ENAME            SAL JOB          AV_SAL
---------- ---------- --------- ----------
ALLEN           1600 SALESMAN       1400
JONES           2975 MANAGER   2758.33333
BLAKE           2850 MANAGER   2758.33333
TURNER          1500 SALESMAN       1400
ADAMS           1100 CLERK        1037.5
MILLER          1300 CLERK        1037.5

已选择 6 行。
```

2. 多行子查询

实例 10-20 执行时首先完成子查询，该子查询的结果就如一个表数据的集合。我们已经知道单行子查询返回一个单行数据，把单行数据作为某种条件来优化或满足查询需求。而多行子查询在现实中也有很多应用场合，下面我们将依次介绍三种常用的多行子查询。

（1）IN

在多行子查询中必须使用多行比较运算符，下面将讲解如何使用 IN 比较符实现多行子查询。IN 比较符返回子查询中的每一个值，一旦有与该值相等的数据行，则输出这些满足条件的数据行，如实例 10-21 所示，查询所在岗位中工资最低的员工信息。

【实例 10-21】查询所在岗位中工资最低的员工信息。

```
SQL> select ename,job,sal,hiredate
  2  from emp
  3  where sal  in (select min(sal)
  4                 from emp
  5                 group by job);

ENAME      JOB                 SAL HIREDATE
-------------------- ------------------ -------------- ------------------
SMITH      CLERK               800   17-12 月-80
MARTIN     SALESMAN           1250   28-9 月 -81
WARD       SALESMAN           1250   22-2 月 -81
KING       PRESIDENT          5000   17-11 月-81
CLARK      MANAGER            2450   09-6 月 -81
FORD       ANALYST            3000   03-12 月-81
SCOTT      ANALYST            3000   19-4 月 -87

已选择 7 行。
```

IN 子查询表 EMP 中按照工作岗位分类的每个岗位的最少工资，查询结果如实例 10-22 所示。

【实例 10-22】查询表 EMP 中按照工作岗位分类的每个岗位的最少工资。

```
SQL> select min(sal)
  2  from emp
  3  group by job;
```

```
  MIN(SAL)
----------
       800
      1250
      5000
      2450
      3000
```

如果将实例 10-21 的条件更换成实例 10-22 的输出结果，会得到相同的结果，如实例 10-23 所示。

【实例 10-23】查询所在岗位中工资最低的员工信息。

```
SQL> select ename,job,sal,hiredate
  2  from emp
  3  where sal  in (800,1250,5000,2450,3000);


ENAME      JOB              SAL HIREDATE
-------------------- ----------------- -------------- -----------------
SMITH      CLERK            800   17-12 月-80
MARTIN     SALESMAN        1250   28-9 月 -81
WARD       SALESMAN        1250   22-2 月 -81
KING       PRESIDENT        5000   17-11 月-81
CLARK      MANAGER         2450   09-6 月 -81
FORD       ANALYST          3000   03-12 月-81
SCOTT      ANALYST          3000   19-4 月 -87

已选择 7 行。
```

（2）ALL

为了说明 ANY 比较符在多行子查询中的作用，我们先看一个实例，如实例 10-24 所示，查询表 EMP 中工资大于或等于所有岗位的最大平均工资的员工信息，即先查询每个岗位的平均工资，而查询条件是比这些平均工资的最大值大或相等的员工信息。

【实例 10-24】查询表 EMP 中工资>=所有岗位的最大平均工资的员工信息。

```
SQL>select ename,job,sal
  2   from emp
  3   where sal >= all(
  4                select avg(sal)
  5                from emp
  6*               group by job)

ENAME      JOB              SAL
------------------ ---------------------- ----------------
KING       PRESIDENT        5000
```

为了分析“>=ALL”，我们通过实例 10-25 执行实例 10-24 的子查询。

【实例 10-25】执行实例 10-24。

```
SQL> select avg(sal)
  2  from emp
  3  group by job;

  AVG(SAL)
----------
    1037.5
      1400
      5000
2758.33333
      3000
```

该实例执行后有 6 行数据输出，最大值为 5000，而实例 10-24 输出了一行数据，该员工的 SAL 值为 5000。“>=ALL”为大于或等于最大的含义。而“=ALL”则没有任何意义，一个值不会和多个值相等。

实例 10-24 的查询结果和如下的查询结果是相同的，如实例 10-26 所示。

【实例 10-26】查询表 EMP 中工资>=5000 的员工信息。

```
SQL> select ename,job,sal
  2  from emp
  3  where sal>=5000;

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
```

（3）ANY

ANY 是任何一个的意思，所以“<ANY”表示比多行返回结果中的任何一个值都小，所以“<ANY”表示小于最大的，而相反“>ANY”表示大于最小的，而“=ANY”表示和任何一个值相等即可，它和 IN 的含义一样。下面我们给出实例来演示 ANY 比较符用于多行子查询的结果。

【实例 10-27】使用 ANY 比较符实现多行子查询。

```
SQL> select ename,job,sal
  2  from emp
  3  where sal > any (
  4                select avg(sal)
  5                from emp
  6                group by job);

ENAME      JOB              SAL
------------------ ------------------------- ----------
KING       PRESIDENT        5000
FORD       ANALYST          3000
SCOTT      ANALYST          3000
JONES      MANAGER         2975
BLAKE      MANAGER         2850
```

```
CLARK      MANAGER        2450
ALLEN      SALESMAN       1600
TURNER     SALESMAN       1500
MILLER     CLERK          1300
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250

ENAME      JOB            SAL
------------------ ------------------------- ----------
ADAMS      CLERK          1100

已选择 12 行。
```

为了验证“>ANY”的作用，我们先执行一个查询，查询按照岗位分类的平均工资，如实例 10-28 所示。

【实例 10-28】查询按照岗位分类的平均工资。

```
SQL> select avg(sal)
  2  from emp
  3  group by job;

  AVG(SAL)
----------
    1037.5
      1400
      5000
2758.33333
      3000
```

上述输出的最小值为 1037.5，我们再使用实例 10-29 来验证“>ANY”的作用。

【实例 10-29】查询表 EMP 中工资大于最小平均岗位工资的员工信息。

```
SQL> select ename,job,sal
  2  from emp
  3  where sal > 1037.5;

ENAME      JOB            SAL
---------- --------- ----------
ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
JONES      MANAGER        2975
MARTIN     SALESMAN       1250
BLAKE      MANAGER        2850
CLARK      MANAGER        2450
SCOTT      ANALYST        3000
KING       PRESIDENT      5000
TURNER     SALESMAN       1500
ADAMS      CLERK          1100
FORD       ANALYST        3000
```

```
ENAME      JOB            SAL
---------- --------- ----------
MILLER     CLERK          1300

已选择 12 行。
```

实例 10-27 和实例 10-29 的查询结果是一样的，不过前者使用 ANY 多行子查询，而后者使用单行子查询。

10.4 连接查询

在数据库的设计中，良好的表设计可以消除数据冗余，在规范化过程中多数的表都满足第三范式，即在满足第一范式的基础上消除了部分依赖和传递依赖，但是再好的设计也不能消除冗余，因为总需要多表查询，所以需要使用外键提供多表查询的“纽带”，Oracle 使用连接实现多表查询。Oracle 提供了多种类型的连接：乘积连接（又名笛卡尔乘积）、相等连接、自然连接、自连接、不等连接和外连接，下面将依次介绍。

10.4.1 乘积连接

笛卡尔积是数据库关系运算中的概念，这里我们不给出严格的关系代数定义，而是只给出解释性的说明，对于维护数据库的 DBA 来讲更具有可操作性。笛卡尔积的结果是两个表中的所有行的一个组合，如果表 1 有 n 行，而表 2 有 m 行，则笛卡尔积的结果有 n*m 行。通过实例 10-30 演示笛卡尔积的计算结果。

【实例 10-30】笛卡尔积的计算查询。

```
SQL> select e.ename,e.sal,e.hiredate,d.dname,d.loc
  2  from emp e,dept d
  3  order by e.hiredate;

ENAME              SAL  HIREDATE       DNAME          LOC
--------------------------- --------- -------------------------- -
SMITH               800  17-12 月-80    ACCOUNTING     NEW YORK
SMITH               800  17-12 月-80    OPERATIONS     BOSTON
…………
ENAME              SAL  HIREDATE       DNAME          LOC
--------------------------- --------- -------------------------- ---
ADAMS             1100  23-5 月 -87     SALES          CHICAGO
TOM               2000  21-7 月 -09     RESEARCH       DALLAS
TOM               2000  21-7 月 -09     SALES          CHICAGO
TOM               2000  21-7 月 -09     OPERATIONS     BOSTON
TOM               2000  21-7 月 -09     ACCOUNTING     NEW YORK

已选择 60 行。
```

实例 10-30 的输出中显示计算结果有 60 行，因为表 EMP 有 15 行记录，而表 DEPT 有 4 行

记录，笛卡尔积是两个表中所有记录组合的结果，所以记录数为 4*15=60 行。我们通过实例 10-31 验证表 EMP、表 DEPT 的记录数和笛卡尔积的记录数。

【实例 10-31】验证表 EMP、表 DEPT 的记录数和笛卡尔积的记录数。

```
SQL> select (select count(*) from emp) "表 EMP 的记录数" , (select count(*)
from dept) "表 DEPT 的记录数 ",
 2  (select count(*) from emp)*(select count(*) from dept) "笛卡尔积的记录数"
 3* from dual

表 EMP 的记录数 表 DEPT 的记录数  笛卡尔积的记录数
------------------------ --------------------------
         15               4             60
```

笛卡尔积不是值得推荐的连接运算，读者可以想象，如果一个数据库中的两个表有近万条记录，显然两个表乘积连接的结果是相当巨大的一个临时表，这样的表不管是对于内存而言，还是 CPU 资源都是很大的冲击，所以在实际使用中最好不要产生笛卡尔积运算。而如下结合的几种连接都是很有效地连接方式。

10.4.2　相等连接

相等连接是使用 WHERE 子句中两个表或多个表中相应列的值的相等条件来选择数据。我们以两个表的相等连接为例，首先找到两个表中满足相等条件的记录（表中相应的行），然后再根据 SELECT 语句选择相应列的值，如实例 10-32 所示。

【实例 10-32】实现相等连接的查询。

```
SQL> select e.ename,d.deptno,d.dname,d.loc
  2  from dept d,emp e
  3  where d.deptno = e.deptno;

ENAME          DEPTNO DNAME          LOC
------------------------- ------------ ------------------------
TOM                20 RESEARCH       DALLAS
SMITH              20 RESEARCH       DALLAS
ALLEN              30 SALES          CHICAGO
WARD               30 SALES          CHICAGO
JONES              20 RESEARCH       DALLAS
MARTIN             30 SALES          CHICAGO
BLAKE              30 SALES          CHICAGO
CLARK              10 ACCOUNTING     NEW YORK
SCOTT              20 RESEARCH       DALLAS
KING               10 ACCOUNTING     NEW YORK
TURNER             30 SALES          CHICAGO

ENAME          DEPTNO DNAME          LOC
------------------------- ------------ ------------------------
ADAMS              20 RESEARCH       DALLAS
JAMES              30 SALES          CHICAGO
```

```
FORD              20 RESEARCH       DALLAS
MILLER            10 ACCOUNTING     NEW YORK

已选择 15 行。
```

上例在表 EMP 和表 DEPT 中查找员工信息，目的是查找每个员工所在的部门号、部门名称和部门所在地（工作地点），但是要求员工所在的部门号和部门表中所在的部门号相等。Oracle 对实例 10-30 的操作过程是：首先通过相等条件连接并取出两个表中相应的行记录，该操作在关系操作中称为选择，然后根据 SELECT 语句中选择的行进行投影操作，最后得到需要的数据输出给用户，在相等连接时，由于相等条件的限制使得连接所涉及的两个表的数据量大幅下降，调入内存的是满足相等条件的记录，而后通过投影操作选择需要的属性。


说 明

使用相等连接时，不可避免地使用 WHERE 子句，而使用 WHERE 子句则需要保证是有效的 WHERE 子句，不然将产生一个乘积连接（笛卡尔积），这样的连接将产生很大的数据量占用大量内存，影响系统的效率。

10.4.3 自然连接

在数据库关系运算中，自然连接是一个重要的连接运算，在 Oracle 数据库中自然连接可以通过相等连接实现，但是为了与数据库关系理论结合起来，对于学过数据库原理的读者来说或许更熟悉自然连接这个概念，所以我们先用自然语言描述什么是自然连接：如果有两个表做自然连接，一般这两个表至少有一个相同的属性，首先将相同属性的相同值的行连接起来，然后去掉一个重复的相同属性的值，如果表 1 有 n 个属性，而表 2 有 m 个属性，二者具有 r 个相同属性，则自然连接的结果是具有 n+m-r 个属性的记录。我们通过实例 10-33 演示自然连接的过程和计算结果。

【实例 10-33】实现自然连接的查询。

```
SQL> select e.empno,e.ename,e.job,e.mgr,e.hiredate,e.sal,e.comm,e.deptno,d.dname,d.loc
  2  from emp e,dept d
  3* where e.deptno = d.deptno

EMPNO ENAME    JOB      MGR HIREDATE      SAL     COMM     DEPTNO DNAME       LOC
---------- ---------- --------- ---------- -------------- ---------- ---
7565 TOM     MANAGER 7839 21-7月 -09      2000     22         20 RESEARCH    DALLAS
…………
已选择 15 行。
```

其实，上述输出只是实例 10-32 的相等连接多了几个属性而已，在实际中由于实现自然连接需要知道两个表的属性信息，如共同属性、属性类型和数量等信息，使用起来没有相等连接便捷，所以很少在数据库维护中使用，一般会使用相等连接代替。

10.4.4 自连接

自连接，顾名思义是一个表与自己连接，通过一些限制性的查询满足查询需要。 在 EMP

表中，我们知道 MANAGER 对于 SALESMAN 具有管理权，而我们需要一张表说明每一个销售员与他的经理之间的关系，则可以使用自连接实现这个查询，如实例 10-34 所示。

【实例 10-34】使用自连接查询销售员与他的经理之间的关系。

```
SQL>  select e1.ename,e1.job,e2.mgr,e2.ename,e2.job
  2  from emp e1,emp e2
  3  where e1.mgr = e2.empno
  4  and e1.job like 'SALESMAN';

ENAME      JOB              MGR ENAME      JOB
----------------- ------------------------- ----------------------------
WARD       SALESMAN        7839 BLAKE      MANAGER
MARTIN     SALESMAN        7839 BLAKE      MANAGER
TURNER     SALESMAN        7839 BLAKE      MANAGER
ALLEN      SALESMAN        7839 BLAKE      MANAGER
```

在表 EMP 中 EMPNO 属性是员工号，而 MGR 属性说明该员工所隶属的上级管理者的员工号，通过实例 10-34 中的条件 where e1.mgr = e2.empno 将一个员工和他的上级管理者连接起来。



上述过程通过分步操作也可以实现，即先创建一个和 EMP 一样的表，然后通过条件限制查询结果，但是这样虽然可以实现查询目的，但是对于新建的表的维护不容易实现，增加了维护表的复杂性，如刚刚创建了 EMP 表的副本，在实现连接操作前原 EMP 表被更新，则这种更新很难即时反映到新创建的表中。

10.4.5　不等连接

不等连接和相等连接对应，即两个表连接操作时，限制条件是不等条件，下面我们查询表 EMP 中每个员工的工资属于哪个等级，并且按照岗位排序，此时我们需要和表 SALGRADE 做不等连接。为了更清楚地理解表 SALGRADE 的内容，先查看该表中的信息，如实例 10-35 所示。

【实例 10-35】查看表 salgrade 的全部数据信息。

```
SQL> conn scott/oracle
已连接。
SQL> select *
  2  from salgrade;

     GRADE      LOSAL      HISAL
---------- ---------- ----------
         1        700       1200
         2       1201       1400
         3       1401       2000
         4       2001       3000
         5       3001       9999
```

表 SALGRADE 有三个列属性，分别是 GRADE 工资等级、LOSAL 某一水平的最低工资和

HISAL 某一水平的最高工资。

下面我们查询 EMP 表中每个员工的工资等级水平，和表 SALGRADE 做不等值连接，如实例 10-36 所示。

【实例 10-36】EMP 表中每个员工的工资等级与表 SALGRADE 的不等值连接。

```
SQL> select e.empno,e.ename,e.job,e.hiredate,s.grade
  2  from emp e,salgrade s
  3  where e.sal between s.losal and s.hisal
  4* order by e.job

     EMPNO ENAME      JOB       HIREDATE             GRADE
------------------ ------------------ ---------------- ------
      7788 SCOTT      ANALYST   19-4 月 -87              4
      7902 FORD       ANALYST   03-12 月-81              4
      7876 ADAMS      CLERK     23-5 月 -87              1
      7369 SMITH      CLERK     17-12 月-80              1
      7900 JAMES      CLERK     03-12 月-81              1
      7934 MILLER     CLERK     23-1 月 -82              2
      7698 BLAKE      MANAGER   01-5 月 -81              4
      7566 JONES      MANAGER   02-4 月 -81              4
      7782 CLARK      MANAGER   09-6 月 -81              4
      7565 TOM        MANAGER   21-7 月 -09              3
      7839 KING       PRESIDENT 17-11 月-81              5

     EMPNO ENAME      JOB       HIREDATE             GRADE
------------------ ------------------ ---------------- ---------
      7654 MARTIN     SALESMAN  28-9 月 -81              2
      7521 WARD       SALESMAN  22-2 月 -81              2
      7499 ALLEN      SALESMAN  20-2 月 -81              3
      7844 TURNER     SALESMAN  08-9 月 -81              3

已选择 15 行。
```

实例 10-36 不等连接与实例 10-32 相等连接的不同在于连接的条件不同，在实例 10-36 中使用了 BETWEEN…AND 数值范围来限制查询条件，是不等查询条件。在上例的结果中每个员工的工资都需要和 SAL 表中的每个等级的工资范围比较，直到找到对应的等级水平。

10.4.6 外连接

为了更直观地说明外连接，我们给出实例 10-37，通过该实例的分析说明外连接的概念与实现方法。

【实例 10-37】外连接查询。

```
SQL> select ename,job,emp.deptno,dept.deptno,dname,loc
  2  from emp ,dept
  3* where emp.deptno(+)= dept.deptno
```

```
ENAME      JOB         DEPTNO     DEPTNO DNAME         LOC
---------- --------- ---------- ---------- -------------- -------------
TOM        MANAGER        20          20 RESEARCH      DALLAS
SMITH      CLERK          20          20 RESEARCH      DALLAS
ALLEN      SALESMAN       30          30 SALES         CHICAGO
WARD       SALESMAN       30          30 SALES         CHICAGO
JONES      MANAGER        20          20 RESEARCH      DALLAS
MARTIN     SALESMAN       30          30 SALES         CHICAGO
BLAKE      MANAGER        30          30 SALES         CHICAGO
CLARK      MANAGER        10          10 ACCOUNTING    NEW YORK
SCOTT      ANALYST        20          20 RESEARCH      DALLAS
KING       PRESIDENT      10          10 ACCOUNTING    NEW YORK
TURNER     SALESMAN       30          30 SALES         CHICAGO

ENAME      JOB         DEPTNO     DEPTNO DNAME         LOC
---------- --------- ---------- ---------- -------------- -------------
ADAMS      CLERK          20          20 RESEARCH      DALLAS
JAMES      CLERK          30          30 SALES         CHICAGO
FORD       ANALYST        20          20 RESEARCH      DALLAS
MILLER     CLERK          10          10 ACCOUNTING    NEW YORK
                                      40 OPERATIONS    BOSTON

已选择 16 行。
```

在上述输出中，没有员工在 DEPTNO 为 40 的部门，但是在表 EMP 和表 DEPT 做等值连接时希望显示表 DEPT 中的 DEPTNO 为 40 的部门信息，使用外连接就可以实现这类查询。



在使用外连接时需要用到符号（+），外连接运算符号必须放在缺少该属性值的一侧。

10.5　本章小结

数据查询是数据库操作中使用很频繁的一类操作，本章的开始处讲解了简单查询，通过简单查询的实现方法和规范，读者可以很容易地掌握并体会查询的方法和过程。在简单查询中我们重点讲解了如何使用别名以及连接运算符、如何使用 DISTINCT 语句和在简单查询中使用算数运算。简单查询都是针对单个表的查询。条件查询和子查询是使用频率较高的查询方式，我们给出 6 种子查询，它们几乎覆盖了子查询的所有方法，读者需要认真学习和体会。

第 11 章

索引与约束

Oracle 使用索引的目的是为了迅速找到需要的数据，当然任何事物都有两面，建立索引可以迅速找到需要的数据，但是在某些情况下也带来性能开销，本章将分析索引的建立和使用维护，并给出使用索引的一些建议。约束定义了一系列的规则，使得用户不会把无效的数据（或称为不合法的数据）存入表中，Oracle 提供了丰富的约束类型，这些类型包括非空约束、唯一约束、主键约束、条件约束和外键约束等。

11.1　索引

索引是 Oracle 的一个对象，索引中存储了特定列的排序数据，实现对表的快速访问。使用索引可以很快查找到建立索引时列的值所在的行，而不必对表实现全表扫描，所以适当的使用索引可以减少磁盘 I/O 量。下面将给出索引的特点总结，这样读者在接下来使用索引时，头脑中就有一个概念。

索引的特点：

- 对于具有只读特性或较少插入、更新或删除操作的大表通常可以提高查询速度。
- 可以对表的一列或多列建立索引。
- 建立索引的数量没有限制。
- 索引需要磁盘存储，需要 Oracle 自动维护。
- 索引对用户而言是透明的，是否使用索引是由 Oracle 决定的。

11.1.1　建立索引

Oracle 使用 CREATE INDEX 指令建立索引，我们通过实例 11-1 说明如何建立索引，此时我们使用 SCOTT 用户的表 EMP 作为实例，假设该表是很大的表，且用户经常使用用户名查询该表中的数据。

【实例 11-1】对 EMP 表建立索引。

```
SQL> conn scott/oracle
已连接。
```

```
SQL> create index emp_ename_idx
  2  on emp(ename);

索引已创建。
```

此时，对表 EMP 的列 ENAME 建立了索引，我们知道索引是需要存储空间的，也就是索引也占用磁盘空间，那么索引存储在哪个表空间，以及如何查看已经建立的索引信息呢？Oracle 使用 USER_INDEXES 数据字典实现，如实例 11-2 所示。

【实例 11-2】使用 USER_INDEXES 数据字典查询索引信息。

```
SQL> col index name for a20
SQL> col index_type for a10
SQL> col table_name for a20
SQL> col tablespace name for a20
SQL> run
  1  select index_name,index_type,table_name,tablespace_name
  2* from user_indexes

INDEX_NAME          INDEX_TYPE  TABLE_NAME           TABLESPACE_NAME
---------------------------- -------------------- ----------
PK_EMP              NORMAL      EMP                  USERS
EMP_ENAME_IDX       NORMAL      EMP                  USERS
PK_DEPT             NORMAL      DEPT                 USERS
```

我们使用数据字典 USER_INDEXES 可以详细地查看当前用户所拥有的索引信息，如上例输出所示，刚刚建立的索引名字 INDEX_NAME 为 EMP_ENAME_IDX，该索引所依赖的表 TABLE_NAME 为 EMP，存储在用户表空间 USERS 中。这样通过数据字典视图 USER_INDEXES 可以很清楚地知道关于当前的所有索引信息。

读者在索引维护中需要知道索引所对应的表空间的信息，因为需要了解该表空间所在的磁盘 I/O 或该磁盘的使用情况来平衡磁盘读写，如实例 11-3 所示。

【实例 11-3】查看索引所对应的表空间信息。

```
SQL> col "索引名" for a15
SQL> col "索引对应的表空间名" for a20
SQL> col "索引对应的磁盘文件" for a40
SQL> select a.index_name "索引名" ,a.tablespace_name
"索引对应的表空间名" ,b.file_name "索引对应的磁盘文件"
  2  from dba indexes a,dba data files b
  3  where a.index_name like 'EMP%'
  4* and a.tablespace_name = b.tablespace_name

索引名          索引对应的表空间名  索引对应的磁盘文件
--------------- -------------------- ----------
EMP_ENAME_IDX   USERS         F:\ORACLE\PRODUCT\10.2.0\ORADATA\LEEJIA\
                                   USERS01.DBF
```

下面再创建一个索引，在表 EMP 的列 ENAME 和 SAL 上创建多列索引，并且指定表空间，

如实例 11-4 所示，首先创建一个索引表空间。

【实例 11-4】创建表空间。

```
SQL> conn system/oracle
已连接。
SQL> create tablespace index_tbs
  2  datafile 'd:/index/index_tbs1.dbf'
  3  size 100M
  4  autoextend on;

表空间已创建。
```

此处创建表空间 INDEX_TBS 的目的就是存放索引，下面演示如何创建对表的多列索引，并且指定表空间，如实例 11-5 所示。

【实例 11-5】创建表的多列索引。

```
SQL> create index emp_ename_sal_idx
  2  on emp(ename,sal)
  3  tablespace index_tbs;

索引已创建。
```

我们依旧使用数据字典 USER_INDEXES 查看是否成功创建索引，如实例 11-6 所示。

【实例 11-6】查看多列索引 EMP_ENAME_SAL_IDX 的信息。

```
SQL> col index_name for a20
SQL> run
  1  select index_name,table_name,tablespace_name
  2  from user indexes
  3* where index_name like 'EMP%'

INDEX_NAME           TABLE_NAME TABLESPACE
------------------------------------ -------------------- -------------
EMP_ENAME_IDX         EMP        USERS
EMP_ENAME_SAL_IDX     EMP        INDEX_TBS
```

从实例 11-6 的输出可以看出索引 MP_ENAME_SAL_IDX 是建立在表 EMP 上的，而且其存储表空间为 INDEX_TBS。但是如何查看一个索引是建立在表的哪几列上呢？答案是使用数据字典 USER_IND_COLUMNS。

在直观理解了如何创建单列索引和多列索引后，我们给出创建索引的语法格式：

```
CREATE [UNIQUE|BITMAP] INDEX [schema.] index_name
ON [schema.]table name
(column_name[DESC]ASC][, column_name[DESC]ASC]]………)
[REVERSE]
[TABLESPACE tablespace name]
[PCTFREE n]
[INITRANS n]
[MAXTRANS n]
```

```
[instorage state]
[LOGGING|NOLOGGING]
[NOSORT]
```

下面解释各个参数的含义。

- UNIQUE：说明该索引是唯一索引。
- BITMAP：创建位图索引。
- DESC|ASC：说明创建的索引为降序或升序排列。
- REVERSE：说明创建反向键索引。
- TABLESPACE：说明要创建的索引所存储的表空间。
- PCTFREE：索引块中预先保留的空间比例。
- INITRANS：每一个索引块中分配的事务数。
- MAXTRANS：每一个索引块中分配的最多事务数。
- instorage state：说明索引中区段 EXTENT 如何分配。
- LOGGING|NOLOGGING：说明要记录（不记录）索引相关的操作，并保存在联机重做日志中。
- NOSORT：不需要在创建索引时按键值进行排序。

11.1.2　查看索引

数据字典 USER_IND_COLUMNS 使得我们可以很方便地查找一个索引所对应的列的信息。在第 11.1.1 节已经建立了两个索引，一个是单列索引 EMP_ENAME_IDX，另一个是多列索引 EMP_ENAME_SAL_IDX。如实例 11-7 所示查询与索引相关的列的信息。

【实例 11-7】查询与索引相关的列信息。

```
SQL> col column_name for a40
SQL> select index_name,table_name,column_name
  2  from user_ind_columns
  3* where index_name like 'EMP%'

INDEX_NAME                TABLE_NAME COLUMN_NAME
----------------------------------- --------------------- ---
EMP_ENAME_IDX              EMP        ENAME
EMP_ENAME_SAL_IDX          EMP        SAL
EMP_ENAME_SAL_IDX          EMP        ENAME
```

实例 11-7 的输出结果说明了索引 EMP_ENAME_SAL_IDX 是基于表 EMP 的两列 SAL 和 ENAME 创建的。

当然数据字典 USER_INDEXES 也可以很方便地查看索引信息，下面再给出一个实例。

【实例 11-8】使用数据字典 USER_INDEXES 查看索引信息。

```
SQL> col dropped for a10
SQL> run
```

```
  1 select index_name,table_name,table_owner,dropped,tablespace_name
  2 from user_indexes
  3* where index_name like 'EMP%'

INDEX_NAME           TABLE_NAME TABLE_OWNE DROPPED    TABLESPACE
------------------------------------- ---------- ----------------
EMP_ENAME_IDX          EMP        SCOTT      NO        USERS
EMP_ENAME_SAL_IDX      EMP        SCOTT      NO        INDEX_TBS
```

实例 11-8 用于查看我们创建的两个索引的信息，以索引 EMP_ENAME_SAL_IDX 为例，该索引对应的表为 EMP，该表所属的用户为 SCOTT，并且 DROPPED 为 NO，其存储表空间为 INDEX_TBS。这里 DROPPED 的含义是：该索引是否是被删除的一个标记。在 Oracle 10g 中，当删除一个对象时，先把该对象放入垃圾箱而不是立即从库中删除，如果删除一个对象，该对象就标记 DROPPED 为 YES。

11.1.3 B-树索引

B-树索引是 Oracle 默认的索引类型，研究 B-树索引也可以帮助理解位图索引和反向键索引。

叶子节点包含索引的实际值和该索引条目的行 ID，即 ROWID。B-树索引的结构如图 11-1 所示。

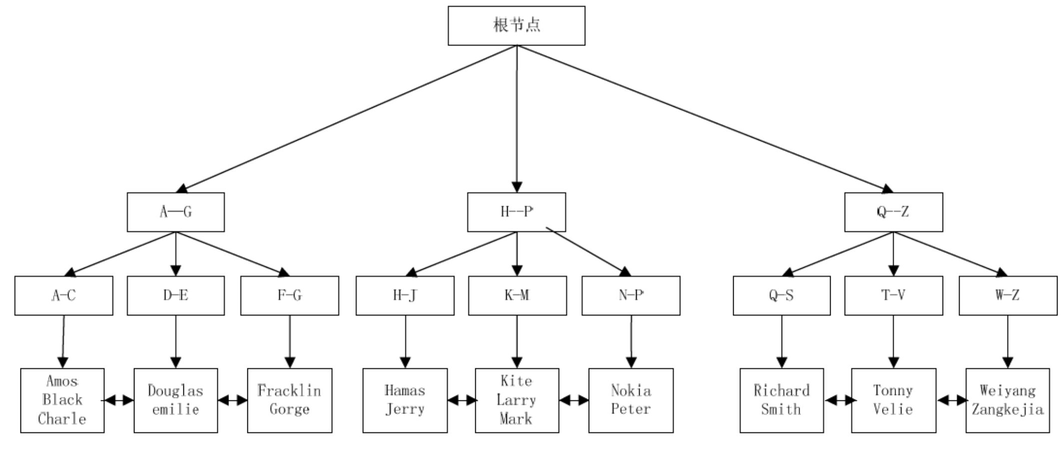


图 11-1　B-树索引结构图

B-树索引结构有三个基本组成部分：根节点、分支节点和叶子节点，其中根节点位于索引结构的最顶端，而叶子节点位于索引结构的最底端，中间为分支节点。

在叶子节点中存储了实际的索引列的值和该列所对应的记录的行 ID，即 ROWID，ROWID 是唯一的 Oracle 指针，指向该行的物理位置，使用 ROWID 是 Oracle 数据库中访问行最快的方法。叶子节点其实是一个双向链表，每个叶子节点包含一个指向下一个和上一个叶子结点的指针，这样在一定范围内建立索引以搜索需要的记录。

每个分支节点又包含其他分支节点，Oracle 设计的 B-树索引结构保证了 B-树索引从根到叶子都有相等的分支节点，保证了 B-树索引的平衡，这样就不会因为基表的数据插入、删除操作

造成 B-树索引变得不平衡，从而影响索引的性能，并且如果一个叶子节点为空，则 Oracle 会释放该空间用于它处。

11.1.4　位图索引

位图索引是 Oracle 10g Enterprise Edition 支持的索引机制。位图索引使用位图标识被索引的列值，它适用于没有大量更新任务的数据仓库，因为使用位图索引时，每个位图索引项与表中大量的行有关联，当表中有大量数据更新、删除和插入时，位图索引相应地需要做大量更改，而且索引所占用的磁盘空间也会明显增加，并且索引在更新时受影响的索引需要锁定，所以位图索引不适合于有大量更新操作的 OLTP 系统，虽然可以通过重建索引类位图索引，但是对于有大量更新操作的表而言最好不选择使用位图索引。

下面以一个 SQL 查询作为实例，解释位图索引的过程，该语句为：

```
SELECT EMPNO,ENAME,SAL
FROM EMP
WHERE JOB = 'SALESMAN';
```

上述查询语句的目的是在 EMP 表中查询工作岗位是 SALESMAN 的员工的员工号、姓名和薪水，此时假设已经在 EMP 表的 JOB 列建立了位图索引，其结构如图 11-2 所示。

Index on JOB

JOB =' CLERK'	1 0 0 0 0 0 0 0 0 0 1 1 0 1
JOB =' SALESMAN'	0 1 1 0 1 0 0 0 0 1 0 0 0 0
JOB =' PRESIDENT'	0 0 0 0 0 0 0 0 1 0 0 0 0 0
JOB =' MANAGER'	0 0 0 1 0 1 1 0 0 0 0 0 0 0
JOB =' ANALYST'	0 0 0 0 0 0 0 1 0 0 0 0 1 0

图 11-2　位图索引结构图

在该索引图中，共有 5 类 JOB，每类 JOB 对应 14 个比特位（对应 14 行记录），其中某行在该列的值与 JOB 值对应，则使用比特 1 表示，如 JOB = 'CLERK'，第一行在该列对应的值是 CLERK，就用比特 1 表示，否则用比特 0 表示，其他 JOB 类似。

图 11-3 是位图索引操作的逻辑视图，通过该视图读者可以更清楚地了解 SQL 语句执行时，如何使用位图索引。

通过位图索引扫描 JOB='CLERK'对应的位图记录，找到值为 1 的行记录，即找到需要查找的数据。

下面给出一个实例来创建位图索引，如实例 11-9 所示。

【实例 11-9】创建位图索引。

```
SQL> create bitmap index emp_job_bitmap_idx
  2  on emp(job);

索引已创建。
```

此时，我们成功创建了位图索引 EMP_ENAME_BITMAP_IDX。该索引基于表 EMP 的 ENAME 列创建。下面通过实例 11-10 查看该索引信息，主要是关注其类型标识。

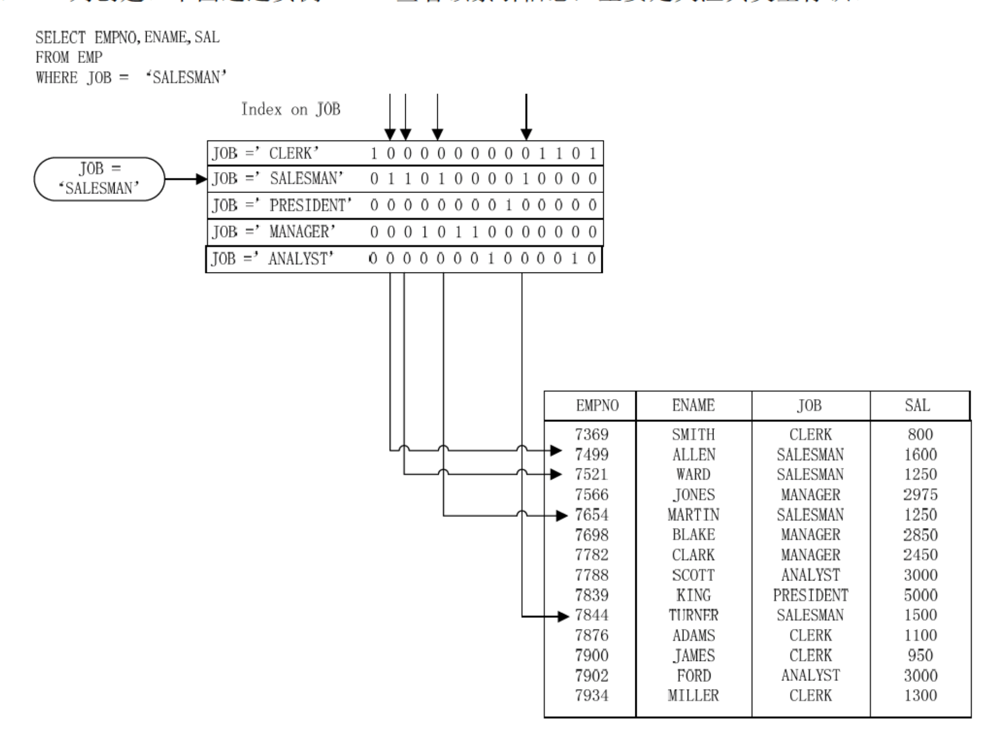


图 11-3 位图索引逻辑结构图

【实例 11-10】查看位图索引。

```
SQL> col index_name for a25
SQL> col index_type for a15
SQL> col table_name for a10
SQL>select index_name,index_type,table_name,status
  2  from user_indexes
  3* where index_name like 'EMP%'

INDEX_NAME                 INDEX_TYPE     TABLE_NAME STATUS
------------------------------------------- -------------------------- --
EMP_JOB_BITMAP_IDX          BITMAP         EMP         VALID
```

实例 11-10 的输出说明了创建的位图索引信息，其中 INDEX_TYPE 为 BITMAP，说明这是一个位图索引。

11.1.5 反向键索引

反向键索引是指在创建索引的过程中对索引列创建的索引键值的字节反向，使用反向键索引的好处是将值连续插入到索引中时反向键能避免争用。

反向键索引适用于一种特殊的情形，如果一个索引值是按照序列值递增的，这样当连续插入大量数据时，所有的记录都将插入 B-树索引结构中的最右侧的叶子节点，并且会写入同一叶子节点中，这样难以避免产生争用问题而影响索引性能。正是为了避免这个问题才引入了反向键索引。使用反向键索引使得每个键值被颠倒顺序，将序列性的键值分散开，使得键值平衡地保存在叶子节点中，如图 11-4 所示为键值颠倒的示意图。

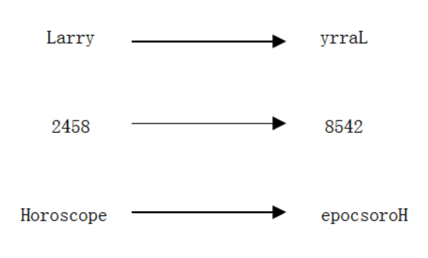


图 11-4　反向键索引的键值

创建反向键索引需要使用 REVERSE 关键字，如实例 11-11 所示。

【实例 11-11】创建反向键索引。

```
SQL> create index emp_sal_reverse_idx
  2  on emp(sal) reverse;

索引已创建。
```

同样，我们通过数据字典 USER_INDEXES 查看刚刚创建的反向键索引信息，如实例 11-12 所示。

【实例 11-12】通过数据字典视图查看反向键索引信息。

```
SQL>  select index_name,index_type,table_name
  2   from user indexes
  3  where index_name like 'EMP%';

INDEX_NAME              INDEX_TYPE             TABLE_NAME
----------------------------------- ------------------
EMP_ENAME_BITMAP_IDX   BITMAP                      EMP
EMP_SAL_REVERSE_IDX     NORMAL/REV                 EMP
```

11.1.6　基于函数的索引

在用户查询数据时，如果查询语句的 WHERE 子句中有函数存在，Oracle 使用函数索引将加快查询速度。基于函数的索引，使用表的列的函数值作为键值建立索引结构，下面说明如何使用 UPPER 函数创建基于函数的索引。

【实例 11-13】创建基于 UPPER 函数的函数索引。

```
SQL> create index dept_dname_idx
  2  on dept(UPPER(dname));

索引已创建。
```

如上所示，我们创建了一个基于表 DEPT 中列 DNAME 的函数索引，创建该索引时首先将列 DNAME 中的值转换成大写，然后对大写的 DNAME 创建索引，放入索引表。这样当用户需

要如下的查询：SELECT UPPER(DNAME) FROM DEPT WHERE UPPER(DNAME) ='NEW YORK'时，Oracle 就不必对 WHERE 子句的条件做转化并逐行检索，对于选择的结果也不必使用 UPPER 函数再做转换的计算，显然此时使用基于函数的索引会极大地提高查询速度，如果该表很大的话，性能的提高是很明显的。

【实例 11-14】通过数据字典 USER_INDEXES 查看基于函数的索引信息。

```
SQL> col index_type for a30
SQL> run
  1  select index_name,index_type,table_name
  2  from user_indexes
  3* where index_name like 'DEPT%'

INDEX_NAME          INDEX_TYPE                     TABLE_NAME
------------------------------------ -------------------------
DEPT_DNAME_IDX       FUNCTION-BASED NORMAL          DEPT
```

上述输出中，索引 DEPT_DNAME_IDX 的类型 INDEX_TYPE 为 FUNCTION_BASED NORMAL，说明它是基于函数的正常索引，该索引是在表 DEPT 上创建的。使用实例 11-15 查看该索引对应列的信息。

【实例 11-15】通过数据字典 USER_IDX_COLUMNS 查看基于函数的索引信息。

```
SQL>  select index_name,table_name,column_name
  2   from user_ind_columns;

INDEX_NAME             TABLE_NAME COLUMN_NAME
---------------------------------------- ------------------- --
PK_DEPT                 DEPT       DEPTNO
PK_EMP                  EMP        EMPNO
EMP_ENAME_BITMAP_IDX   EMP        ENAME
EMP_SAL_REVERSE_IDX     EMP        SAL
DEPT_DNAME_IDX          DEPT       SYS_NC00004$
```

从实例 11-15 的输出可以看出，索引 DEPT_DNAME_IDX 的 COLUMN_NAME 是系统赋予的一个值，因为这个列不是 Oracle 可以使用明确的列名标识的，所以 Oracle 就使用自己的方法来标识 GOLUMN_NAME 列名。

11.1.7 本地分区索引

如果一个基表是分布在多个分区上的，即该表是分区表，则对表建立的索引和表分区一一对应，即每个分区表的索引和该分区表在同一个分区上存储，这样在数据分区和索引分区之间存在对应关系，这就是本地分区索引的特点。

使用本地分区索引使得索引和该索引所对应的数据分区分布在同一个分区中，使得索引和基表得以均匀分布，如果修改了基表分区，则 Oracle 自动维护对应的索引分区。创建本地分区索引如实例 11-16 所示，我们需要对一个分区表建立本地分区索引，对于普通表则没有意义。这里

假设表 products 为分区表，对该表的列 PRO_DATE 生产日期建立索引。

【实例 11-16】创建本地分区索引。

```
SQL> create index products pro date idx
  2  on products(pro_date) local
  3  tablespace product_tbs;
```

和本地分区索引对应的是全局分区索引，全局分区索引可以分区，也可以不分区，索引和基表分区之间不存在一一对应关系。对于全局分区索引的创建和使用这里不再详解，对于初学者只要掌握概念就可以了，随着学习地深入和数据库的需求，读者可以参看 Oracle 10g 的 CONCEPT 文档或其他书籍。

11.1.8　监控索引

虽然创建了索引，但是如何监控索引的使用情况呢？否则就无法判断创建的索引的有效性。对于无效的索引可以删除以释放磁盘空间并较少 DML 操作带来的修改索引的各种开销。

下面通过一个实例来说明如何监控索引的使用，该方法在 Oracle 9i 以上都适用，首先我们确定要监控索引 EMP_ENAME_BITMAP_IDX 是否被使用，此时需要执行如下的操作以启动监控行为。

【实例 11-17】启动对索引 EMP_ENAME_BITMAP_IDX 的监控。

```
SQL> alter index EMP_ENAME_BITMAP_IDX
  2  monitoring usage;

索引已更改。
```

接下来，我们要等待一个周期，在这个过程中等待用户对表 EMP 的各种操作，对于 OLTP（联机事务处理）系统而言，这个周期可能很短，而对于数据仓而言，则需要更多的时间来监控。

下面我们模拟这个周期中的两个查询。

```
SQL> select empno,ename,sal
  2  from emp
  3  where ename like 'S%';

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7788 SCOTT            3000

SQL> select ename
  2  from emp;

ENAME
----------
ADAMS
ALLEN
```

```
BLAKE
.........
WARD

已选择 14 行。
```

然后，我们终止对索引 EMP_ENAME_BITMAP_IDX 的监控，如实例 11-18 所示。

【实例 11-18】终止监控 EMP_ENAME_BITMAP_IDX 索引。

```
SQL> alter index EMP ENAME BITMAP IDX
  2  nomonitoring usage;

索引已更改。
```

现在，就可以使用数据字典视图 v$object_usage 查看 EMP_ENAME_BITMAP_IDX 索引的使用情况。

```
SQL> select index_name, table_name, monitoring, used
  2  from v$object usage;

INDEX_NAME             TABLE_NAME MON USE
-------------------------------------- -------------------
EMP_ENAME_BITMAP_IDX  EMP         NO  YES
索引已更改。
```

输出说明索引 EMP_ENAME_BITMAP_IDX 是基于表 EMP 创建的，当前没有监控该索引，因为 MON 为 NO，该索引已经被 Oracle 使用过了，因为 USED 为 YES。

在数据字典视图 v$object_usage 中，还提供了 START_MONITORING 和 END_MONITORING 来说明某个索引的监控周期，即起始时间和结束时间，如实例 11-19 所示。

【实例 11-19】查看索引 EMP_ENAME_BITMAP_IDX 的监控周期。

```
SQL> select index_name,start_monitoring,end_monitoring
  2  from v$object_usage
  3  where index name = 'EMP ENAME BITMAP IDX';

INDEX_NAME             START_MONITORING    END_MONITORING
---------------------------------------- ----------------
EMP_ENAME_BITMAP_IDX  08/08/2009 20:42:16       08/08/2009 20:43:28
```



Oracle 也提供其他工具来监控索引使用的有效性，如使用 EXPLAIN PLAN 的输出和使用 SQL trace 等工具，有兴趣的读者可以参考相关书籍，这里不再详述。

11.1.9 重建索引

索引需要维护，不然如果建立了索引的表中有大量的删除和插入操作，会使得索引很大，因为在删除操作后，删除值所占用的索引空间不能被索引自动重新使用，而插入操作会不断使得索引变大，对于大表和 DML 操作很频繁的表索引的维护是很重要的，Oracle 提供了一个 REBUILD 指令来

重建索引，使得索引空间可以重用删除值所占用的空间，使得索引更加紧凑，如实例 11-20 所示。

【实例 11-20】重建索引。

```
SQL> alter index emp_ename_bitmap_idx
  2  rebuild;

索引已更改。
```



说 明

使用索引重建不影响用户使用索引，但是有一些限制条件，如不能使用 DDL 操作和 DML 操作。

在重建索引时也可以使用其他参数代表要重建的索引的表空间，要求读者必须事先创建了表空间 INDEX_TBS1，如实例 11-21 所示。

【实例 11-21】重建索引并迁移其表空间。

```
SQL> alter index dept_dname_idx
  2  rebuild
  3  tablespace index_tbs1

索引已更改。
```

为了验证重建索引 DEPT_DNAME_IDX 的结果，我们给出实例 11-22。

【实例 11-22】使用数据字典 USER_INDEXES 验证重建索引的有效性及表空间的迁移变化。

```
SQL> select index_name,table_name,tablespace_name ,status
  2  from user_indexes;

INDEX_NAME             TABLE_NAME TABLESPACE_NAME      STATUS
--------------------------------------- ------------------- -------------
PK_EMP                     EMP        USERS                  VALID
EMP_ENAME_BITMAP_IDX  EMP        USERS                  VALID
EMP_SAL_REVERSE_IDX    EMP        USERS                  VALID
PK_DEPT                    DEPT       USERS                  VALID
DEPT_DNAME_IDX          EMP        INDEX_TBS1             VALID
```

通过查询输出可以看出索引 DEPT_DNAME_IDX 的状态 STATUS 为 VALID，说明该索引重建后是有效的，可以使用。而 TABLESPACE_NAME 为 INDEX_TBS1，说明已经成功迁移了索引 DEPT_DNAME_IDX 到表空间 INDEX_TBS1。

在重建索引时，还可以修改索引的其他参数，如 PCTFREE 或者 STORAGE 子句等，如实例 11-23 所示。

【实例 11-23】重建索引 EMP_ENAME_BITMAP_IDX 并修改存储参数。

```
SQL> alter index EMP_ENAME_BITMAP_IDX
  2  rebuild
  3  pctfree 30
  4  storage (next 100k);
```

```
索引已更改。
```

也可以使用联机重建索引的方式，利用这样的方式重建索引时，可以执行 DML 操作，但是不能使用 DDL 操作，如实例 11-24 所示。

【实例 11-24】联机重建索引。

```
SQL> alter index dept dname idx
  2  rebuild online;

索引已更改。
```

11.1.10 维护索引

维护索引就是修改索引的各种参数，在维护索引前我们先需要知道当前索引的参数设置，如实例 11-25 所示。

【实例 11-25】查询当前索引的参数设置。

```
SQL> col index_name for a20
SQL> select index_name,pct_free,pct_increase,initial_extent,next_extent
  2* from user indexes

INDEX_NAME          PCT_FREE  PCT_INCREASE  INITIAL_EXTENT  NEXT_EXTENT
-------------------------------- -------------- ---------------------- ---
PK EMP                   10                     65536
EMP_SAL_REVERSE_IDX     10                     65536
EMP_JOB_BITMAP_IDX       10                     65536
PK_DEPT                  10                     65536
```

从输出可以看出索引 EMP_JOB_BITMAP_IDX 的 PCT_FREE 为 10%，INITIAL_EXTENT 为 65536 字节。下面我们通过实例 11-26 演示如何通过 REBUILD 修改参数。

【实例 11-26】通过 REBUILD 修改索引参数。

```
SQL> alter index emp_job_bitmap_idx
  2  rebuild
  3  pctfree 30
  4  storage (next 100k);
```

下面再查看修改结果，如实例 11-27 所示。

【实例 11-27】查询修改后索引 EMP_JOB_BITMAP_IDX 的参数。

```
SQL> select index_name,pct_free,pct_increase,initial_extent,next_extent
  2  from user indexes
  3  where index_name = 'EMP_JOB_BITMAP_IDX';

INDEX NAME          PCT FREE  PCT INCREASE  INITIAL EXTENT  NEXT EXTENT
-------------------------------- ------------- --------------------- ------
EMP_JOB_BITMAP_IDX  30                     65536
```

```
索引已更改。
```

此时，索引 EMP_JOB_BITMAP_IDX 的 PCT_FREE 参数已修改为 30%，但是注意 NEXT_EXTENT 仍然是默认值，没有修改。因为索引 EMP_JOB_BITMAP_IDX 存储在表空间 USERS 中，而该表空间是本地管理的表空间索引，无法修改 NEXT_EXTENT 参数。

接下来再演示如何为索引手工分配磁盘空间，如实例 11-28 所示。

【实例 11-28】手工增加索引磁盘空间。

```
SQL> alter index EMP_JOB_BITMAP_IDX
  2  allocate extent;

索引已更改。
```

Oracle 对于每个索引默认的 EXTENT 区段数为 1，此时为索引 EMP_JOB_BITMAP_IDX 增加了一个区段，所以该索引应该有两个区段。我们通过实例 11-29 查询该索引的参数信息。

【实例 11-29】查询索引 EMP_JOB_BITMAP_IDX 的区段信息。

```
SQL> select segment_name,segment_type,tablespace_name,extents
  2  from user segments
  3  where segment_type = 'INDEX'
  4  and segment_name like 'EMP%';

SEGMENT_NAME       SEGMENT_TYPE     TABLESPACE_NAME       EXTENTS
---------------------------------- ----------------------------- -------
EMP_JOB_BITMAP_IDX     INDEX            USERS                        2
EMP_SAL_REVERSE_IDX    INDEX            USERS                        1
```

从上述输出可以看出，索引 EMP_JOB_BITMAP_IDX 的区段 EXTENTS 数量为 2，说明我们已经成功为其添加区段。

另一种重要的维护索引的方式是合并索引碎片，其实合并碎片也是维护磁盘空间的方式。下面将演示如何合并索引碎片，如实例 11-30 所示。

【实例 11-30】合并索引碎片。

```
SQL> alter index emp job bitmap idx coalesce;

索引已更改。
```

通过合并索引碎片可以释放部分磁盘空间，是索引维护的一个重要方法。

11.1.11　删除索引

如果经过索引监控发现一个索引无效，或者出于效率考虑暂时不需要，可以删除该索引，删除索引时可使用 DROP INDEX 指令，如实例 11-31 所示。

【实例 11-31】删除索引。

```
SQL> drop index dept_dname_idx;
```

```
索引已删除。
```

在删除该索引后，为了确认删除结果，我们再使用实例 11-32 通过数据字典 USER_INDEXES 查询删除结果。

【实例 11-32】查看是否删除索引 DEPT_DNAME_IDX。

```
SQL> select index_name,index_type,table_name
  2  from user_indexes
  3  where index_name like 'DEPT%';

未选定行
```

显然我们要查找的索引不存在，说明已经成功删除了索引 DEPT_DNAME_IDX。

11.2 约束

Oracle 引入约束的目的是保证插入表的数据满足一定的要求，这个要求可以理解为业务规则的声明，约束作为数据定义的一部分，所以它是声明性的，而不是过程性的，这种简单的业务规则便于编写代码，而且也便于维护。

使用约束比在应用程序中使用规则验证更有效，执行速度更快。一旦声明了约束，在插入数据、更新数据时 Oracle 将自动使用这些约束来验证数据的有效性。约束的定义即可在 CREATE TABLE 过程的数据定义中定义，也可以在表创建后使用 ALTER TABLE ADD 命令添加必要的索引。同时 Oracle 提供了两类数据字典视图，可以方便地查询索引信息，即*_indexes 和*_cons_columns。关于这两类数据字典的用法在介绍索引类型时会有很多示例，这里不做介绍，其中“*”号表示可以是 USER、DBA 或者 ALL。

约束按照功能可以分为 5 种。

- 非空约束（NOT NULL）：不允许某列为 NULL。
- 主键约束（PRIMARY KEY）：主键约束用于标识主键的唯一性，且不能为 NULL。
- 唯一约束（UNIQUE KEY）：说明该列的值是唯一的，但可以是 NULL。
- 条件约束（CHECK）：说明某列的值必须满足一定的条件，条件约束是最灵活的一种约束类型。
- 外键约束（FOREIGN KEY）：维护从表和主表之间的引用完整性。

11.2.1 非空约束

例如在 SCOTT 用户的员工表 EMP 中，员工号是不允许为 NULL 的，即要求员工号必须存在且为数字类型 NUMER，在表 EMP 的定义过程中已经定义了非空约束。我们通过实例 11-33 查看表 EMP 的 EMPNO 列的约束信息。

【实例 11-33】查看表 EMP 的 EMPNO 列的约束信息。

```
SQL> desc emp;
 名称                                        是否为空?  类型
 -----------------------------------------------------------
 EMPNO                                      NOT NULL   NUMBER(4)
 ENAME                                                 VARCHAR2(10)
 JOB                                                   VARCHAR2(9)
 MGR                                                   NUMBER(4)
 HIREDATE                                              DATE
 SAL                                                   NUMBER(7,2)
 COMM                                                  NUMBER(7,2)
 DEPTNO                                                NUMBER(2)
```

从输出可以注意到列 EMPNO，列值不允许为空（NOT NULL），且其数据类型为 NUMBER(4)，数据类型的验证是 Oracle 默认的验证过程，如果插入的值非空但数据类型不对，也不允许插入数据。

现在向表 EMP 中插入一行数据，此时将 EMPNO 列对应的值设置为 NULL，如实例 11-34 所示。

【实例 11-34】向 EMP 表中插入 EMPNO 为空的记录。

```
SQL> insert into emp (empno,ename,job,sal)
  2  values ('','Tom','SALESMAN',2000);
values ('','Tom','SALESMAN',2000)
        *
第 2 行出现错误:
ORA-01400: 无法将 NULL 插入 ("SCOTT"."EMP"."EMPNO")
```

可见 Oracle 不允许在 SCOTT 用户的表 EMP 的 EMPNO 列中插入空值 NULL。现在我们再插入一行非空记录，如实例 11-35 所示。

【实例 11-35】向 EMP 表中插入一行 EMPNO 非空的记录。

```
SQL> insert into emp (empno,ename,job,sal)
  2  values ('7599','tom','SALESMAN',2000);

已创建 1 行。
```

因为表 EMP 除了 EMPNO 外，其他列是允许插入空值的，只要 EMPNO 列不违反非空约束就可以插入数据，而且上例中，没有说明的列 Oracle 都自动插入空值 NULL。通过实例 11-36 查询表 EMP 中刚插入的记录信息。

【实例 11-36】查询刚插入的表信息。

```
SQL> select empno,ename,job,sal,hiredate,mgr,deptno
  2  from emp
  3  where empno = 7599;

EMPNO ENAME      JOB          SAL  HIREDATE        MGR     DEPTNO
---------- -------------------- -------------------- --------- -----------
7599     tom        SALESMAN     2000
```

从输出可以看出，列 HIREDATE、MGR 和 DEPTNO 都为空值。

当更新一个表时，也不能违反空值约束，更新刚才插入的记录，如实例 11-37 所示。

【实例 11-37】更新表 EMP 中 EMPNO 列为空值。

```
SQL> update emp
  2  set empno = null
  3  where ename ='tom';
set empno = null
    *
第 2 行出现错误:
ORA-01407: 无法更新 ("SCOTT"."EMP"."EMPNO") 为 NULL
```

显然无法将用户 tom 的 EMPNO 设置为 NULL，因为违反了 EMPNO 不能为空值的约束，即违反了非空约束。

下面说明如何在创建了一个表时建立某列的约束，为此，创建一个表 EMP_TEST，如实例 11-38 所示。

【实例 11-38】创建测试表 EMP_TEST。

```
SQL> create table emp_test
  2  ( empno number(4) not null,
  3    ename varchar2(20) not null,
  4    sal   number(6),
  5    hiredate date);

表已创建。
```

下面验证是否成功创建了表 EMP_TEST，如实例 11-39 所示。

【实例 11-39】验证是否成功创建表 EMP_TEST。

```
SQL> desc emp_test;
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                     NOT NULL VARCHAR2(20)
 SAL                                                NUMBER(6)
 HIREDATE                                           DATE
```

表 EMP_TEST 对 ENAME 列和 EMPNO 列设置非空约束，此时我们也可以通过数据字典 USER_CONSTRAINTS 查询相关信息，如实例 11-40 所示。

【实例 11-40】使用数据字典 USER_CONSTRAINTS 查询约束信息。

```
SQL> col constraint_name for a20
SQL> col constraint_type for a20
SQL> col table_name for a20
SQL> select constraint_name,constraint_type,table_name
  2  from user_constraints
  3* where table_name like 'EMP%'
```

```
CONSTRAINT_NAME      CONSTRAINT_TYPE      TABLE_NAME
------------------------------------- -----------------------------------
PK_EMP                     P                  EMP
FK_DEPTNO                  R                  EMP
SYS_C005534                  C                  EMP_TEST
SYS_C005535                  C                  EMP_TEST
```

我们看到在 CONSTRIANT_TYPE 约束类型上表 EMP_TEST 对应的两个约束都是 C 标识，而 CONSTRAINT_NAME 约束名分别为 SYS_C005534 和 SYS_C005535。

首先对于约束名，如果用户没有明确定义约束名，Oracle 将自动生成 SYS_C 为前缀加上 6 位十进制整数组成的约束名。当然，用户最好定义自己的约束名，这样也便于维护。

对于约束类型，Oracle 提供了 4 种类型标识，如下所示。

- P：表示主键约束（PRIMARY KEY）。
- R：表示引用约束（REFERENTIAL INTEGRITY）。
- C：表示条件（CHECK）约束或非空约束（NOT NULL）。
- U：表示唯一约束（UNIQUE）。

下面再看另一个数据字典 USER_CONS_COLUMNS。它可以查询约束建立在哪个表的哪个列上，如实例 11-41 所示。

【实例 11-41】使用数据字典 USER_CONS_COLUMNS 查询约束信息。

```
SQL> col owner for a10
SQL> col column name for a20
SQL> run
  1  select owner,constraint_name,table_name,column_name
  2  from user_cons_columns
  3* where table name ='EMP TEST'

OWNER      CONSTRAINT_NAME      TABLE_NAME           COLUMN_NAME
-------------------- -------------------------------------- -----------
SCOTT         SYS_C005534              EMP_TEST             EMPNO
SCOTT         SYS_C005535            EMP_TEST              ENAME
```

从上例的输出可以清楚地看出，表 EMP_TEST 的非空约束分别建立在列 EMPNO 和列 ENAME 上，而且该约束属于 SCOTT 用户。

11.2.2　唯一约束

唯一约束要求列的值在表中是唯一的，但是可以插入空值 NULL。下面依然使用表 EMP_TEST 来创建唯一约束，首先删除表 EMP_TEST 上对列 ENAME 的非空约束，如实例 11-42 所示。

【实例 11-42】删除表 EMP_TEST 上对列 ENAME 的非空约束。

```
SQL> alter table emp_test
```

```
  2  drop constraint sys_c005535;

表已更改。
```

然后，我们使用指令 ALTER TABLE 对表的 ENAME 列创建一个唯一约束，说明表 EMP_TEST 中的列 ENAME 的值不能重复（NULL 值除外，允许多次插入 NULL 值），如实例 11-43 所示。

【实例 11-43】对表 EMP_TEST 的列 ENAME 创建唯一索引。

```
SQL> alter table emp_test
  2  add constraint emp_test_ename_uk unique(ename);

表已更改。
```

下面通过数据字典 USER_CONSTRAINTS 验证创建结果，如实例 11-44 所示。

【实例 11-44】查询唯一索引 EMP_TEST_ENAME_UK 信息。

```
SQL> select constraint_name,constraint_type,table_name
  2  from user_constraints
  3  where table_name like 'EMP%';

CONSTRAINT_NAME      CONSTRAINT_TYPE      TABLE_NAME
--------------------------------------- ------------
PK_EMP                  P                   EMP
FK_DEPTNO               R                   EMP
SYS_C005534              C                   EMP_TEST
EMP_TEST_ENAME_UK    U                   EMP_TEST
```

我们看到约束 EMP_TEST_ENAME_UK 的类型 CONSTRAINT_TYPE 为 U ，说明它是唯一约束。此时也说明成功创建了唯一约束。

如果读者感兴趣，可以使用数据字典 USER_CONS_COLUMNS 验证约束所属的用户和建立约束的列，这里不再给出。

为了验证唯一约束的“唯一”性，我们向表中插入数据，如实例 11-45 所示。

【实例 11-45】向表 EMP_TEST 中插入记录。

```
SQL> insert into emp_test
  2  values (1,'tom',2000,sysdate);

已创建 1 行。
```

为了确认插入效果，可查询该表中的数据，如实例 11-46 所示。

【实例 11-46】查询表 EMP_TEST 中的数据。

```
SQL> select *
  2  from emp_test;

     EMPNO ENAME                   SAL HIREDATE
-------------------- -------------------------------------------
```

```
          1 tom                          2000 09-8月 -09
```

现在向表中再插入一行数据，注意此时的 ENAME 仍然是'tom'，如实例 11-47 所示。

【实例 11-47】验证表 EMP_TEST 的唯一约束。

```
SQL> insert into emp test
  2  values (2,'tom',3000,sysdate);
insert into emp_test
*
第 1 行出现错误:
ORA-00001: 违反唯一约束条件 (SCOTT.EMP_TEST_ENAME_UK)
```

上述输出清楚地说明了插入操作违反了唯一约束条件，即 SCOTT 用户的 EMP_TEST_ENAME_UK 约束。

那么如果插入空值会出现什么情况呢？我们先前说过唯一约束允许插入空值，因为空值不是任何值，一个 NULL 和另一个 NULL 也不同，如实例 11-48 所示。

【实例 11-48】向表 EMP_TEST 中插入 ENAME 为 NULL 的记录。

```
SQL>  insert into emp_test
  2   values (3,null,3000,sysdate);

已创建 1 行。

SQL>  insert into emp_test
  2   values (2,'',3000,sysdate);

已创建 1 行。
```

实例说明唯一约束是允许插入空值的，因为空值不是任何值，也不和任何值相等，它不违反唯一约束条件。我们查询表 EMP_TEST 中的数据，查看插入数据的结果，如实例 11-49 所示。

【实例 11-49】查看表 EMP_TEST 中的空值记录。

```
SQL> select *
  2  from emp_test;

    EMPNO ENAME                     SAL  HIREDATE
----------------- ---------------------------------------------- -
          1   tom                   2000  09-8月 -09
          3                         3000  09-8月 -09
          2                         3000  09-8月 -09
```

11.2.3　主键约束

主键约束和唯一键约束很相似，但是主键约束不允许插入空值 NULL。主键唯一标识一个表中的记录，主键可以在一列或多列上创建。

为了说明如何创建主键约束，我们为表 EMP_TEST 创建基于列 EMPNO 的主键约束。希望每一个员工号能唯一标识一个员工记录，如实例 11-50 所示。

【实例 11-50】创建表 EMP_TEST 上基于列 EMPNO 的主键约束。

```
SQL> alter table emp_test
  2 add constraint emp_test_empno_pk
  3 primary key(empno);

表已更改。
```

接着为了确认创建结果，我们使用实例 11-51 来进行验证。

【实例 11-51】查询主键约束 EMP_TEST_EMPNO_PK 的信息。

```
SQL>  select owner,constraint_name,table_name,column_name
  2  from user_cons_columns
  3  where table_name = 'EMP_TEST';

OWNER      CONSTRAINT NAME     TABLE NAME        COLUMN NAME
-------------------- ----------------------------------- ---------
SCOTT      SYS_C005534            EMP_TEST             EMPNO
SCOTT      EMP_TEST_ENAME_UK    EMP_TEST             ENAME
SCOTT      EMP_TEST_EMPNO_PK    EMP_TEST             EMPNO
```

我们看到主键约束 EMP_TEST_EMPNO_PK 创建成功，且该约束属于用户 SCOTT，是基于表 EMP_TEST 的 ENAME 列创建的。

我们已经说过主键约束要求唯一且不能插入空值。此时通过实例 11-52 验证是否可以插入重复值。

【实例 11-52】向表 EMP_TEST 中插入含重复值的记录来验证主键约束。

```
SQL> insert into emp_test
  2 values (3,'larry',3000,sysdate);
insert into emp_test
*
第 1 行出现错误:
ORA-00001: 违反唯一约束条件 (SCOTT.EMP_TEST_EMPNO_PK)
```

显然上述结果提示违反了唯一约束条件，所以不能插入重复的值，那是否可以插入空值呢？我们用实例 11-53 来验证。

【实例 11-53】向表 EMP_TEST 中插入含空值的记录来验证主键约束。

```
SQL> insert into emp_test
  2 values (null,'lay',3000,sysdate);
values (null,'lay',3000,sysdate)
        *
第 2 行出现错误:
ORA-01400: 无法将 NULL 插入 ("SCOTT"."EMP_TEST"."EMPNO")
```

此时，说明空值 NULL 也无法插入表 EMP_TEST 的 EMPNO 列中。

11.2.4　条件约束

条件约束是比较灵活的一类约束，可以根据需要对表设置更多的规则限制，条件约束说明表中每一行的某列或几列的数据必须满足的条件。条件约束可以使用一个或多个约束条件，可以在单一条件约束中使用复合条件，也可以对同一列使用多个条件约束。

条件约束的判断标准是条件约束的返回值，TRUE 表示满足条件，FALSE 表示不满足条件，拒绝执行。条件约束可以在创建表时创建，也可以在表创建完成后对某列创建。首先说明如何在创建表时创建条件约束。我们创建一个表 EMP_TEST2，该表中的 SAL 不允许低于 1000（最低工资保障），如实例 11-54 所示。

【实例 11-54】在创建表的过程中创建条件约束。

```
SQL> create table emp_test2
  2  (empno    number(4) unique,
  3  ename     varchar2(20) not null,
  4  sal       number(6),
  5  hiredate  date,
  6  constraint emp_test2_sal_ck
  7  check(sal>1000));

表已创建。
```

我们再查看是否成功创建条件约束 EMP_TEST2_SAL_CK，如实例 11-55 所示。

【实例 11-55】查看是否成功创建表 EMP_TEST2 的条件约束。

```
SQL> select owner,constraint_name,constraint_type,table_name
  2 from user_constraints
  3 where table name = 'EMP TEST2';

OWNER    CONSTRAINT_NAME     CONSTRAINT_TYPE   TABLE_NAME
---------- --------------------- --------------------- ------------
SCOTT      SYS_C005539            C                  EMP_TEST2
SCOTT      EMP_TEST2_SAL_CK       C                  EMP_TEST2
SCOTT      SYS_C005541            U                  EMP_TEST2
```

输出说明表 EMP_TEST2 具有一个条件约束 EMP_TEST2_SAL_CK、一个非空约束 SYS_C005539 和一个唯一约束 SYS_C005541。

下面说明如何在表创建后，使用 ALTER TABLE 指令创建条件约束，如实例 11-56 所示。

【实例 11-56】对表 EMP_TEST 的 SAL 列创建条件约束。

```
SQL> alter table emp_test
  2 add constraint emp_test_sal_ck
  3 check(sal>1000);

表已更改。
```

同样验证更改结果，看是否对列 SAL 创建了条件约束，如实例 11-57 所示。

【实例 11-57】通过数据 USER_CONSTRAINTS 验证表 EMP_TEST 的条件约束。

```
SQL> select owner,constraint_name,constraint_type,table_name
  2  from user_constraints
  3  where table_name = 'EMP_TEST';

OWNER      CONSTRAINT_NAME      CONSTRAINT_TYPE      TABLE_NAME
---------- ------------------------------------------ ----------
SCOTT      SYS_C005534            C                    EMP_TEST
SCOTT      EMP_TEST_ENAME_UK    U                    EMP_TEST
SCOTT      EMP_TEST_EMPNO_PK    P                    EMP_TEST
SCOTT      EMP_TEST_SAL_CK       C                    EMP_TEST
```

从输出可见约束 EMP_TEST_SAL_CK 已创建成功，而且 CONSTRAINT_TYPE 为 C（条件约束）。那么如果插入一条违反条件的记录会出现什么情况呢？如实例 11-58 所示。

【实例 11-58】验证表 EMP_TEST 中的条件查询。

```
SQL> insert into emp_test
  2  values (4,'lay',500,sysdate)
  3  ;
insert into emp_test
*
第 1 行出现错误:
ORA-02290: 违反检查约束条件 (SCOTT.EMP_TEST_SAL_CK)
```

显然无法插入数据，因为插入的记录中 SAL 为 500<1000，所以该行记录违反了条件查询的约束条件，Oracle 拒绝插入该记录。

11.2.5 外键约束

外键约束不仅仅涉及一个表，它还涉及主表和从表，使用外键约束使得 Oracle 可以维护主表和从表之间的引用完整性。简单地说，如果要插入从表中的一行数据，则某列的值要参考主表中引用参考的值，而不能无限制地插入记录。

我们先通过实例说明外键约束的作用。这里使用 SCOTT 用户的表 EMP 和 DEPT，表 EMP 中的 DEPTNO 需要参考 DEPT 中的 DEPTNO，即表 EMP 中的 DEPTNO 为外键，它参考表 DEPT 中的 DEPTNO。

我们现在向表 EMP 中插入一行数据，其中 DEPTNO = 50，查看是否可以成功，如实例 11-59 所示。

【实例 11-59】向表 EMP 中插入 DEPTNO=50 的记录来验证外键约束。

```
SQL> insert into emp (empno,ename,job,sal,deptno)
  2  values (7598,'tom','SALESMAN',3000,50);
insert into emp (empno,ename,job,sal,deptno)
*
第 1 行出现错误:
ORA-02291: 违反完整约束条件 (SCOTT.FK_DEPTNO) - 未找到父项关键字
```

显然没有插入成功，提示违反了完整性的约束条件，其实就是违反了外键约束条件，因为表 EMP 中的 DEPTNO 要参考 DEPT 表中的 DEPTNO，如果表 DEPT 中没有这个部门，在表 EMP 中根本无法插入不存在部门的一个员工记录，其实就是要求 EMP 表中的部门必须存在于表 DEPT 中。这样就维护了表 DEPT 和表 EMP 之间的 DEPTNO 数据的完整性。此时读者应该可以体会外键约束的作用了。

下面创建表 EMP_TEST3，目的是演示如何创建外键索引，以及出现的问题，如实例 11-60 所示。

【实例 11-60】创建表 EMP_TEST3。

```
 SQL>  create table emp test3
  2   (empno number(4) unique,
  3   ename  varchar2(20) not null,
  4   sal   number(6),
  5   hiredate  date,
  6   deptno   number(4),
  7   constraint emp_test3_deptno_fk
  8  foreign key (deptno) references dept(deptno));

表已创建。
```

此时，我们创建了一个表 EMP_TEST3，该表有三个约束，一个是对列 EMPNO 的唯一约束，另一个是对列 ENAME 的非空约束，最后一个是对列 DEPTNO 的外键约束，它参考表 DEPT 的 DEPTNO 列的值。

同样使用数据字典 USER_CONSTRAINTS 来验证上述约束信息，如实例 11-61 所示。

【实例 11-61】查询表 EMP_TEST3 的约束信息。

```
SQL>select constraint_name,constraint_type,table_name
  2  from user_constraints
  3* where table name = 'EMP TEST3'

CONSTRAINT_NAME      CONSTRAINT_TYPE      TABLE_NAME
------------------------------------- ----------
SYS C005542              C                      EMP TEST3
SYS_C005543              U                      EMP_TEST3
EMP_TEST3_DEPTNO_FK    R                      EMP_TEST3
```

上述输出说明创建了非空索引、唯一索引，二者的名字都是由系统提供的，外键索引 EMP_TEST3_DEPTNO_FK 也成功创建，其 CONSTRAINT_TYPE 为 R 也说明了其类型。

下面插入数据到 EMP_TEST3，如实例 11-62 所示。

【实例 11-62】向表 EMP_TEST3 中插入数据。

```
SQL> insert into emp_test3
  2  values (1000,'tom',2000, sysdate,20);

已创建 1 行。
```

```
SQL> insert into emp_test3
  2  values (1001,'larry',3000,sysdate,30);

已创建 1 行。
```

我们成功向表 EMP_TEST3 中插入两行数据，且没有违反任何约束条件。读者可以自行分析。下面再插入一行数据，其中 DEPTNO 的值为 60，如实例 11-63 所示。

【实例 11-63】向表 EMP_TEST3 中插入一行 DEPTNO=60 的记录。

```
SQL> insert into emp_test3
  2  values (1001,'larry',3000,sysdate,30);

已创建 1 行。

SQL> insert into emp_test3
  2  values (1002,'lay',3000,sysdate,60);
insert into emp_test3
*
第 1 行出现错误:
ORA-02291: 违反完整约束条件 (SCOTT.EMP_TEST3_DEPTNO_FK) - 未找到父项关键字
```

此时，提示违反了完整性约束条件，没找到父关键字，说明因为外键约束的存在使得插入数据记录时，DEPTNO 的值必须存在于表 DEPT 中，下面通过实例 11-64 说明外键约束的信息。

【实例 11-64】查看外键约束 EMP_TEST3_DEPTNO_FK 的信息。

```
SQL>select owner,constraint_name,constraint_type,table_name,
r_constraint_name
  2  from user_constraints
  3* where table_name ='EMP_TEST3'

OWNER   CONSTRAINT NAME CONSTRAINT TYPE TABLE NAME     R CONSTRAINT NAME
---------------- ------------------------------------- -----------
SCOTT      SYS_C005542            C                    EMP_TEST3
SCOTT      SYS_C005543            U                    EMP_TEST3
SCOTT      EMP_TEST3_DEPTNO_FK  R       EMP_TEST3             PK_DEPT
```

从上述输出可以看出，表 EMP_TEST3 的外键约束 EMP_TEST3_DEPTNO_FK 的引用约束为 PK_DEPT，从名字可以知道该索引是表 DEPT 的主键索引。

下面查看表 DEPT 中的数据，观察 DEPTNO 列的值，如实例 11-65 所示。

【实例 11-65】查询表 DEPT。

```
SQL> select *
  2  from dept;

   DEPTNO DNAME          LOC
------------------- ------------------------ -------------
       10 ACCOUNTING     NEW YORK
```

```
          20  RESEARCH        DALLAS
          30  SALES            CHICAGO
          40  OPERATIONS       BOSTON
```

我们看到表 DEPT 中 DEPTNO 列的值为 10、20、30 和 40，在表 EMP_TEST3 中插入记录时，记录对应的 DEPTNO 列的值必须在上述 4 个值的范围内。

11.2.6　删除约束

如果不需要约束，可以随时删除该约束，要求用户具有删除约束的条件并知道约束的名字。一般可以通过数据字典 USER_CONSTRAINTS 或 DBA_CONSTRAINTS 来查找约束的名字，进而使用 DROP CONSTRAINT 指令删除约束，如实例 11-66 所示为删除表 EMP_TEST3 中的唯一约束 SYS_C005543。

【实例 11-66】删除表 EMP_TEST3 的唯一约束。

```
SQL> alter table emp_test3
  2  drop constraint SYS_C005543;

表已更改。
```

为了验证删除结果，再查询表 EMP_TEST3 中的约束信息，如实例 11-67 所示。

【实例 11-67】查询是否删除表 EMP_TEST3 中的唯一约束 SYS_C005543。

```
SQL> select constraint_name,constraint_type,table_name
  2  from user_constraints
  3  where table_name = 'EMP_TEST3';

CONSTRAINT_NAME      CONSTRAINT_TYPE      TABLE_NAME
------------------------------------------------------------------------------
SYS_C005542                 C                   EMP_TEST3
EMP_TEST3_DEPTNO_FK    R                  EMP_TEST3
```

输出结果说明已经成功删除了唯一约束 SYS_C005543。

再删除表 DEPT 中的主键约束 PK_DEPT，如实例 11-68 所示。

【实例 11-68】删除表 DEPT 中的主键索引 PK_DEPT。

```
SQL> alter table dept
  2  drop constraint pk_dept;
drop constraint pk_dept
              *
第 2 行出现错误:
ORA-02273: 此唯一/主键已被某些外键引用
```

Oracle 拒绝了删除该约束操作，提示“此唯一/主键已被某些外键引用”，说明该主键被其他表作为外键引用，不能删除，那该怎么办？Oracle 提供了 CASCADE 关键字可以解决找个问题，如实例 11-69 所示。

【实例 11-69】使用 CASCADE 关键字删除表 DEPT 的主键索引。

```
SQL> alter table dept
  2  drop constraint pk_dept cascade;

表已更改
```

使用 CASCADE 关键字强制切断了主表和从表之间的外键引用关系，从而可以成功删除。

11.2.7 其他约束维护

使用 DISABLE 指令关闭约束，该指令既可以使用在 CREATE TABLE 中，也可以用在 ALTER TABLE 指令中。我们先查看表 EMP_TEST3 的约束状态，如实例 11-70 所示。

【实例 11-70】查看表 EMP_TEST3 的约束信息。

```
SQL> select constraint_name,constraint_type,table_name,status
  2  from user_constraints
  3* where table_name = 'EMP_TEST3'

CONSTRAINT_NAME    CONSTRAINT_TYPE  TABLE_NAME           STATUS
----------------------------------- ------------------------------ ----
SYS_C005542              C                 EMP_TEST3            ENABLED
EMP_TEST3_DEPTNO_FK  R                   EMP_TEST3            ENABLED
```

我们看到表 EMP_TEST3 中的两个约束状态 STATUS 都处于启动 ENABEL 状态。

下面使用实例 11-71 演示如何关闭约束。

【实例 11-71】关闭表 EMP_TEST3 的约束 SYS_C005542。

```
SQL> alter table emp_test3
  2  disable constraint sys_c005542;

表已更改。
```

下面查看该约束的当前状态是否为关闭状态，如实例 11-72 所示。

【实例 11-72】查看表 EMP_TEST3 的约束 SYS_C005542 的状态。

```
SQL> select constraint_name,constraint_type,table_name,status
  2  from user_constraints
  3  where table_name = 'EMP_TEST3'
  4  and constraint name like 'SYS%';

CONSTRAINT_NAME  CONSTRAINT_TYPE      TABLE_NAME      STATUS
-------------------------------- ------------------------------------ ----
SYS_C005542            C                     EMP_TEST3            DISABLED
```

此时，我们成功关闭了约束，在需要的时候可以使用 ALTER TABLE…ENABLE 指令启动关闭的约束。

再关闭表 DEPT 的主键约束 PK_DEPT，如实例 11-73 所示。

【实例 11-73】关闭表 DEPT 的主键约束 PK_DEPT。

```
SQL> alter table dept
  2  disable constraint pk_dept;
alter table dept
*
第 1 行出现错误:
ORA-02297: 无法禁用约束条件 (SCOTT.PK_DEPT) - 存在相关性
```

我们看到此时无法禁用表 DEPT 的主键约束，因为表 EMP_TEST3 和表 EMP 都引用表 DEPT 的主键作为外键，所以此时必须使用 CASCADE 指令关闭存在有引用完整性关系的约束，如实例 11-74。

【实例 11-74】关闭存在引用完整性关系的约束。

```
SQL>  alter table dept
  2   disable constraint pk_dept cascade;

表已更改。
```

此时表已经更改成功，再次查看表 DEPT 的约束 PK_DEPT 的状态信息，如实例 11-75 所示。

【实例 11-75】查看表 DEPT 中约束 PK_DEPT 的状态信息。

```
SQL> select constraint_name,constraint_type,table_name,status
  2  from user_constraints
  3  where table name = 'DEPT'
  4  ;

CONSTRAINT_NAME     CONSTRAINT_TYPE    TABLE_NAME     STATUS
------------------------------------ ------------------------------ ----
PK_DEPT               P                  DEPT                DISABLED
```

此时，表 DEPT 的约束 PK_DEPT 的状态 STATUS 已经是 DISABLED 了。

下面再启动表 DEPT 的主键约束，毕竟还有两个表在使用其作为外键约束，如实例 11-76 所示。

【实例 11-76】启用表 DEPT 的主键索引 PK_DEPT。

```
SQL> alter table dept
  2  enable constraint pk_dept;

表已更改。
```

11.3　本章小结

本章重点介绍了索引和约束的概念，索引在一定条件下可以提高查询速度，但是对于不同的应用环境，应该使用不同的索引类型，Oracle 提供 B-树索引、位图索引、反向键索引、基于函

数的索引、本地分区索引和全局分区索引。其中 B-树索引和位图索引是常用的索引类型，对于索引的维护本章介绍了重建和维护索引，合理有效地索引维护可提高索引使用的效率，减少索引占用的磁盘空间。

Oracle 主要提供了 5 种约束，即非空约束、唯一约束、主键约束、条件约束和外键约束，每一类约束都有自己的特点，用户只要根据自己的商业系统定制相应的约束即可，约束的定义是描述性的，它比在应用程序中使用数据过滤规则效率要高很多。约束可以根据需要关闭、启用或者删除，对于有完整性关系的约束需要知道使用 CASCADE 关键字来切断引用完整性关系。

第 12 章

数据库的启动与关闭

数据库的启动涉及一系列的文件读取和数据一致性检查等操作，但数据库启动时会首先启动数据库实例（Instance），在这个过程中数据库获得一些内存空间，并启动了必须的后台监控进程，而后会读取控制文件再进一步打开各种数据文件，最后完成数据库的启动任务。数据库的关闭执行了和启动相反的过程。下面将详细介绍数据库的启动和关闭过程，以及在启动和关闭过程中涉及的各类文件。

12.1　启动数据库

启动数据库需要读者以 DBA 用户的身份登录，只有 DBA 用户才具有打开和关闭数据库的权限。启动数据库时，如果具有相应的权限，只需要输入 startup 指令就可以打开数据库。但是如果权限不够，则会出现如下错误提示，如实例 12-1 所示。

【实例 12-1】权限不足时启动数据库。

```
SQL> startup
ORA-01031: insufficient privileges
```

如果用户具有 DBA 权限，在输入 start 指令后 Oracle 数据系统要执行一系列的复杂操作，如读取参数文件、控制文件、打开数据文件，进行数据一致性检验等，下面详细介绍数据库的启动过程。

12.1.1　数据库的启动过程

数据库的启动过程涉及到三个状态，在每个状态中数据库都做不同的事情，同时这三个状态适用于数据库的不同维护要求。这三个状态如下。

- NOMOUNT 状态：只打开了数据库实例。
- MOUNT 状态： Oracle 根据参数文件中控制文件的位置找到并打开控制文件，读取控制文件中的各种参数信息，如数据文件和日志文件的位置等。
- OPEN 状态：数据库将打开数据文件并进行一系列的检查工作，这些检查工作用于数据恢复。

图 12-1 中给出了一个数据库启动流程图，当数据库启动到 NOMOUNT 状态时，此时 Oracle 只打开数据库实例。在启动到 MOUNT 状态时，打开实例并读取控制文件。启动到 OPEN 状态则打开数据文件、日志文件等各类必须的数据库文件。在启动数据库时，我们可以直接启动数据库到 OPEN 状态，即打开数据库。但是这个过程仍然经历了我们介绍的三个状态过程，即：NOMOUNT→MOUNT→OPEN。在接下来的几节中我们将分别介绍这三个状态。

图 12-1 说明了数据库启动到不同状态的过程和涉及的操作。1 表示数据库启动到 NOMOUNT 状态，2 表示数据库启动到 MOUNT 状态，3 表示数据库启动到 OPEN 状态，方框表示启动到每一种状态涉及的操作，启动到 MOUNT 状态必然经历了 NOMOUNT 状态，启动到 OPEN 状态必然经历了 NOMOUNT 和 MOUNT 状态。

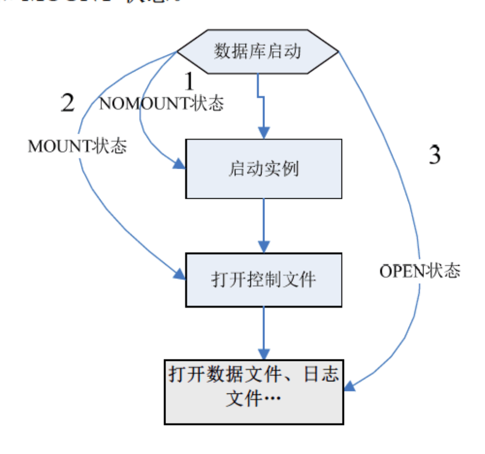


图 12-1　数据库启动流程图及相关状态

12.1.2　数据库启动到 NOMOUNT 状态

启动数据库到 NOMOUNT 状态时，会首先搜索参数文件，读者只需要知道参数文件中存放了一些参数信息，如数据库缓冲区大小、重做日志缓冲区大小等。根据这些参数分配内存，即 SGA，然后启动必须的后台进程，有 5 个后台进程是必须启动的，它们是 DBWR（数据库写进程）、LGWR（日志写进程）、SMON（系统监控进程）、PMON（进程监控进程）、CKPT（检验点进程）。

该过程不涉及控制文件和数据文件，只需要一个参数文件就可以启动到 NOMOUNT 状态。启动到 NOMOUNT 状态的指令如实例 12-2 所示。

【实例 12-2】启动到 NOMOUNT 状态。

```
C:\Documents and Settings\Administrator>sqlplus /nolog

SQL*Plus: Release 11.1.0.6.0 - Production on 星期五 10月 16 12:21:44 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.
```

```
SQL> connect /as sysdba
已连接到空闲例程。
SQL> startup nomount;
ORACLE 例程已经启动。

Total System Global Area  535662592 bytes
Fixed Size                  1334380 bytes
Variable Size             281019284 bytes
Database Buffers          247463936 bytes
Redo Buffers                5844992 bytes
```

数据库的启动过程记录在告警追踪文件中，该告警追踪文件中包括了数据库启动的信息，它存放在参数 BACKGROUND_DUMP_DEST 定义的目录下，告警日志文件的名字为 alert_orcl.log，如果用户不知道，可以使用如下指令查询告警日志的存储目录，如实例 12-3 所示。

【实例 12-3】查看存储告警追踪文件的参数值。

```
SQL> show parameter background_dump_dest;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
background_dump_dest                 string      f:\app\administrator\diag\rdbm
                                                 s\orcl\orcl\trace
```

上例的输出可以说明，记录告警追踪文件位于参数 background_dump_dest 指定的目录下，即在笔者的计算机上，告警追踪文件存储在 f:\app\administrator\diag\rdbm s\orcl\orcl\trace 下，如图 12-2 所示。

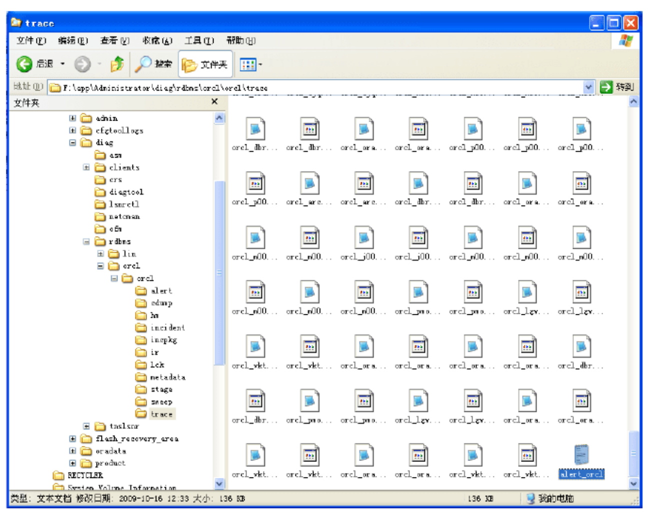


图 12-2　Windows 平台上告警日志文件的存储目录

在上图的目录下，由于笔者的 SID 为 lin，所以告警文件名为 linALRT.log。下面我们查看一下该告警文件的内容，以确认数据库启动到 NOMOUNT 状态时的启动详细过程。下面的内容是从 linALRT.log 拷贝的，这部分记录说明了数据启动到 NOMOUNT 状态的启动过程。由于告警

追踪文件记录太长，我们将分开说明。

启动数据库实例的前期工作，如归档目录、启动审计等，代码如下：

```
Fri Oct 16 12:22:15 2009
Starting ORACLE instance (normal)
LICENSE_MAX_SESSION = 0
LICENSE_SESSIONS_WARNING = 0
Picked latch-free SCN scheme 2
Using LOG_ARCHIVE_DEST_1 parameter default value as
F:\app\Administrator\product\11.1.0\db_1\RDBMS
Using LOG_ARCHIVE_DEST_10 parameter default value as
 USE_DB_RECOVERY_FILE_DEST
Autotune of undo retention is turned on.
IMODE=BR
ILAT =18
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 11.1.0.6.0.
```

下面这部分记录内容是用于读取参数文件获得系统参数值，这些参数包括分配的SGA中非自动管理的内存组件大小，如大池和流池、控制文件的位置、数据库块大小以及还原表空间名的。

```
Using parameter settings in server-side spfile
F:\APP\ADMINISTRATOR\PRODUCT\11.1.0\DB_1\DATABASE\SPFILEORCL.ORA
System parameters with non-default values:
  processes                = 150
  large_pool_size           = 52M
  streams_pool_size         = 12M
  memory_target             = 808M
  control_files             =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL"
  control_files             =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL"
  control_files             =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL"
  db_block_size             = 8192
  compatible               = "11.1.0.0.0"
  db_recovery_file_dest     = "F:\app\Administrator\flash_recovery_area"
  db_recovery_file_dest_size= 2G
  undo_tablespace           = "UNDOTBS1"
  remote_login_passwordfile= "EXCLUSIVE"
  db_domain                = ""
  dispatchers               = "(PROTOCOL=TCP) (SERVICE=orclXDB)"
  audit_file_dest           = "F:\APP\ADMINISTRATOR\ADMIN\ORCL\ADUMP"
  audit_trail               = "DB"
  db_name                  = "orcl"
  open_cursors              = 300
  diagnostic_dest           = "F:\APP\ADMINISTRATOR"
```

下面这部分记录内容用于启动后台的数据库管理进程：

```
Fri Oct 16 12:22:16 2009
PMON started with pid=2, OS id=4044
Fri Oct 16 12:22:16 2009
VKTM started with pid=3, OS id=1228 at elevated priority
VKTM running at (20)ms precision
Fri Oct 16 12:22:16 2009
DIAG started with pid=4, OS id=2340
Fri Oct 16 12:22:16 2009
DBRM started with pid=5, OS id=3680
Fri Oct 16 12:22:16 2009
PSP0 started with pid=6, OS id=1504
Fri Oct 16 12:22:16 2009
DSKM started with pid=7, OS id=3040
Fri Oct 16 12:22:16 2009
DIA0 started with pid=8, OS id=1584
Fri Oct 16 12:22:16 2009
MMAN started with pid=7, OS id=3900
Fri Oct 16 12:22:16 2009
DBW0 started with pid=9, OS id=2832
Fri Oct 16 12:22:16 2009
LGWR started with pid=10, OS id=4036
Fri Oct 16 12:22:16 2009
CKPT started with pid=11, OS id=2652
Fri Oct 16 12:22:16 2009
SMON started with pid=12, OS id=3172
Fri Oct 16 12:22:16 2009
RECO started with pid=13, OS id=2804
Fri Oct 16 12:22:16 2009
MMON started with pid=14, OS id=1184
starting up 1 dispatcher(s) for network address
'(ADDRESS=(PARTIAL=YES)(PROTOCOL=TCP))'...
Fri Oct 16 12:22:16 2009
MMNL started with pid=15, OS id=2676
starting up 1 shared server(s) ...
ORACLE_BASE from environment = F:\app\Administrator
```

上述文件内容主要说明了数据库启动到 NOMOUNT 状态时的启动详细信息，首先 Oracle 读取参数文件分配内存区，读取其他如数据库块大小等参数，然后启动后台进程，但是没有打开控制文件的信息。

在数据库启动到 NOMOUNT 状态时，并不打开控制文件。在 Oracle 中查看控制文件存储目录的方法是使用视图 v$controlfile，这是一个动态视图，如果数据控制文件没有打开，则无法通过该动态视图查询到控制文件的存储目录。实例 12-4 用于测试在启动到 NOMOUNT 状态时，控制文件是否打开。

【实例 12-4】测试在启动到 NOMOUNT 状态时控制文件是否打开。

```
SQL> select *
  2  from v$controlfile;

未选定行
```

上例说明数据库没有打开控制文件。但是在 NOMOUNT 状态下可以通过参数文件获得控制文件的位置，因为我们知道此时参数文件已经打开。在 NOMOUNT 状态，我们使用 v$parameter 动态视图获得控制文件的位置，如实例 12-5 所示。

【实例 12-5】在 NOMOUNT 状态下使用 v$parameter 视图获得控制文件的位置。

```
SQL> show parameter control_files;

NAME                 TYPE       VALUE
------------------------------ ----------- ------------------------------
control_files          string      F:\APP\ADMINISTRATOR\ORADATA\O
RCL\CONTROL01.CTL, F:\APP\ADMI
                                   NISTRATOR\ORADATA\ORCL\CONTROL
                                   02.CTL, F:\APP\ADMINISTRATOR\O
RADATA\ORCL\CONTROL03.CTL
```

上例的输出说明，当前数据库的控制文件都位于 F:\APP\ADMINISTRATOR\ORADATA\ORCL 目录下。

12.1.3 数据库启动到 MOUNT 状态

数据库在启动到 MOUNT 状态时有两种方式，一是可以直接启动数据库到 MOUNT 状态；二是如果数据库已经启动到 NOMOUNT 状态，然后使用指令 alter database mount 把数据库切换到 MOUNT 状态，本例我们采用后者，在第 12.1.2 节数据库启动到 NOMOUNT 状态后，再启动到 MOUNT 状态，如实例 12-6 所示。

【实例 12-6】在 NOMOUNT 状态下将数据库启动到 MOUNT 状态。

```
SQL> ALTER database mount;

数据库已更改。
```

此时，数据库处于 MOUNT 状态，我们可以通过动态视图 v$controlfile 获得控制文件的存储目录。因为从 NOMOUNT 状态切换到 MOUNT 状态就是 Oracle 打开控制文件的过程。实例 12-7 用于在 MOUNT 状态下查看控制文件的存储目录。

【实例 12-7】在 MOUNT 状态下查看控制文件的存储目录。

```
SQL> col name for a50
SQL> select status,name,block_size
  2  from v$controlfile;
```

```
STATUS  NAME                                                        BLOCK_SIZE
------- ------------------------------------------------------------------------
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL            16384
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL            16384
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL            16384
```

因为我们打开了控制文件，所以可以通过动态数据字典视图 v$controlfile 来查看控制文件的状态、存储目录信息、控制文件本身的数据块大小等信息。在实例 12-7 中查询到的控制文件存储在 F:\APP\ADMINISTRATOR\ORADATA\ORCL 目录下，并且每个控制文件的数据块大小都为 16384 字节。

然后，我们在告警追踪文件中可以查看从 NOMOUNT 状态切换到 MOUNT 状态的过程。

```
Fri Oct 16 12:30:06 2009
alter database mount
Setting recovery target incarnation to 2
Fri Oct 16 12:30:10 2009
Successful mount of redo thread 1, with mount id 1228202830
Database mounted in Exclusive Mode
Lost write protection disabled
Completed: alter database mount。
```

记录显示在时间 Fri Oct 16 12:30:06 2009 执行了“alter database mount”，即将数据库状态转换到 MOUNT 状态的指令。

但是，此时数据库并没有打开，所以数据文件无法读取，我们用实例 12-8 说明在 MOUNT 状态下，数据库没有打开。

【实例 12-8】在 MOUNT 状态下读取数据文件。

```
SQL> select *
  2  from scott.dept;
from scott.dept
          *
ERROR 位于第 2 行:
ORA-01219: 数据库未打开: 仅允许在固定表/视图中查询
```

12.1.4　数据库启动到 OPEN 状态

启动数据库到 OPEN 状态，Oracle 需要检验数据文件的头信息，进行点计数器检查和 SCN 检查来完成实例恢复，至于检查点计数器和 SCN 在控制文件与数据库启动一章中将详细介绍。

数据库启动到 OPEN 状态，有两种方式，一是直接启动到 OPEN 状态，使用指令 startup open 或 startup 来实现（startup 的默认启动方式是启动到 OPEN 状态）。二是如果数据库处于 NOMOUNT 或 MOUNT 状态，可以使用指令 alter database open 切换到 OPEN 状态。这里采用第二种方式，如实例 12-9 所示。

【实例 12-9】在数据库启动到 MOUNT 时打开数据库。

```
SQL> alter database open;
```

```
数据库已更改。
```

数据库处于 OPEN 状态，此时我们可以查询数据库中的表数据，用户 SCOTT 的表都存储在 USERS 表空间中，如实例 12-10 所示。

【实例 12-10】在打开数据库时查询数据库中的表。

```
SQL> select *
  2  from scott.dept;

DEPTNO   DNAME           LOC
--------------- ---------------------------- -------------
       10  ACCOUNTING     NEW YORK
       20  RESEARCH       DALLAS
       30  SALES          CHICAGO
       40  OPERATIONS     BOSTON
```

显然，由于数据库切换到 OPEN 状态，所以打开数据文件时，用户可以操作数据库中的数据文件中的各种对象，如查询指定用户表中的数据。

我们再来看看告警追踪文件 alert_orcl 中记录的该过程：

```
Fri Oct 16 13:19:15 2009
alter database open
Fri Oct 16 13:19:15 2009
LGWR: STARTING ARCH PROCESSES     //启动归档进程。
Fri Oct 16 13:19:15 2009
ARC0 started with pid=19, OS id=4072
Fri Oct 16 13:19:15 2009
ARC1 started with pid=20, OS id=4076
Fri Oct 16 13:19:15 2009
ARC2 started with pid=21, OS id=4080
ARC0: Archival started
ARC1: Archival started
Fri Oct 16 13:19:15 2009
ARC3 started with pid=22, OS id=4084
ARC2: Archival started
ARC3: Archival started
LGWR: STARTING ARCH PROCESSES COMPLETE
Thread 1 opened at log sequence 19      //读取重做日志文件。
ARC0: Becoming the 'no FAL' ARCH
  Current log# 1 seq# 19 mem# 0:
F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
Successful open of redo thread 1
MTTR advisory is disabled because FAST_START_MTTR_TARGET is not set
ARC0: Becoming the 'no SRL' ARCH
ARC2: Becoming the heartbeat ARCH
Fri Oct 16 13:19:16 2009
SMON: enabling cache recovery      //系统监控进程完成恢复任务。
```

```
Successfully onlined Undo Tablespace 2.
Verifying file header compatibility for 11g tablespace encryption..
Verifying 11g file header compatibility for tablespace encryption completed
SMON: enabling tx recovery
Database Characterset is ZHS16GBK
Opening with internal Resource Manager plan
Starting background process FBDA
Starting background process SMCO
Fri Oct 16 13:19:18 2009
FBDA started with pid=23, OS id=352
Fri Oct 16 13:19:18 2009
SMCO started with pid=24, OS id=1504
replication_dependency_tracking turned off (no async multimaster replication found)
Starting background process QMNC
Fri Oct 16 13:19:20 2009
QMNC started with pid=25, OS id=872
Fri Oct 16 13:19:28 2009
db_recovery_file_dest_size of 2048 MB is 8.09% used. This is a
user-specified limit on the amount of space that will be used by this
database for recovery-related files, and does not reflect the amount of
space available in the underlying filesystem or ASM diskgroup.
Fri Oct 16 13:19:38 2009
Completed: alter database open
```

12.2　关闭数据库

关闭数据库同启动数据库的顺序正好相反，其基本思路是首先关闭各种数据文件，关闭打开的控制文件，然后关闭实例，数据库的启动经历了 NOMOUNT、MOUNT 和 OPEN 三个阶段，数据库的关闭正好相反，它经历了 CLOSE、DISMOUNT 和 SHUTDOWN，如图 12-3 所示。

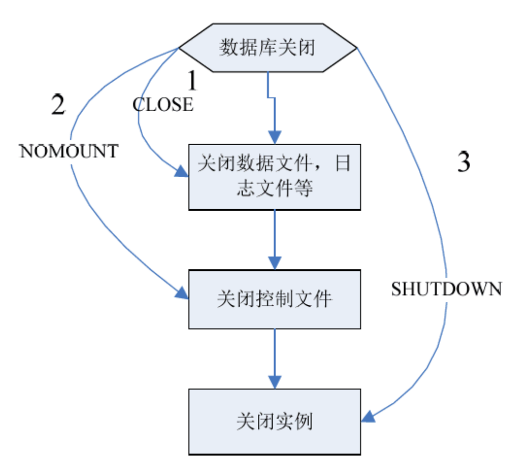


图 12-3　数据库关闭过程和涉及的操作

12.2.1 数据库的关闭过程

正如启动数据库一样，我们可以分步启动数据库，即从 NOMOUNT->MOUNT->OPEN，而关闭数据库也可以分步关闭，即从 CLOSE->DISMOUNT->SHUTDOWN。下面我们分步介绍关闭数据库的过程，此时假设数据库已经正常启动并可以访问。

1. CLOSE 数据库

我们用实例 12-11 说明如何 CLOSE 数据库。

【实例 12-11】关闭数据库。

```
SQL> ALTER DATABASE close;

数据库已更改。
```

此时，可以通过查看告警文件的方式，观察这个过程，如下所示：

```
Fri Oct 16 13:23:47 2009
Starting background process CJQ0
Fri Oct 16 13:23:47 2009
CJQ0 started with pid=29, OS id=884
Fri Oct 16 13:30:32 2009
alter database close
Fri Oct 16 13:30:32 2009
SMON: disabling tx recovery
Stopping background process QMNC
Fri Oct 16 13:30:32 2009
Stopping background process CJQ0
Stopping background process FBDA
Stopping background process SMCO
SMON: disabling cache recovery
Fri Oct 16 13:30:36 2009
Shutting down archive processes
Archiving is disabled
Fri Oct 16 13:30:36 2009
ARCH shutting down
Fri Oct 16 13:30:36 2009
ARCH shutting down
ARC1: Archival stopped
ARC3: Archival stopped
Fri Oct 16 13:30:36 2009
ARCH shutting down
ARC0: Archival stopped
Fri Oct 16 13:30:36 2009
ARCH shutting down
ARC2: Archival stopped
Thread 1 closed at log sequence 19
Successful close of redo thread 1
Completed: alter database close
```

2. DISMOUNT 数据库

我们用实例 12-12 说明如何 DISMOUNT 数据库。

【实例 12-12】将数据库切换到 DISMOUNT 状态。

```
SQL> ALTER DATABASE dismount;

数据库已更改。
```

在告警文件中会有如下记录：

```
Fri Oct 16 13:34:15 2009
alter database dismount
Completed: alter database dismount
```

3. SHUTDOWN 数据库

SHUTDOWN 数据库会关闭数据库实例，使用 SHUTDOWN 关闭数据库时，Oracle 会做一些处理工作，如断开连接、回滚数据等，这些操作依赖于 SHUTDOWN 所选择的参数，下面通过实例 12-13 说明如何 SHUTDOWN 数据库。

【实例 12-13】SHUTDOWN 数据库。

```
SQL> shutdown
ORA-01507: 未安装数据库


ORACLE 例程已经关闭。
```

如果读者的数据库是英文环境，则输出如下：

```
SQL> shutdown
ORA-01507: database not mounted


ORACLE instance shut down。
```

此时，查看告警文件，Oracle 记录的 SHUTDOWN 操作记录如下：

```
Shutting down instance: further logons disabled
Stopping background process MMNL
Stopping background process MMON
Shutting down instance (normal)
License high water mark = 2
Waiting for dispatcher 'D000' to shutdown
All dispatchers and shared servers shutdown
ALTER DATABASE CLOSE NORMAL
ORA-1507 signalled during: ALTER DATABASE CLOSE NORMAL...
ARCH: Archival disabled due to shutdown: 1090
Shutting down archive processes
Archiving is disabled
Archive process shutdown avoided: 0 active
```

```
ARCH: Archival disabled due to shutdown: 1090
Shutting down archive processes
Archiving is disabled
```

12.2.2 数据库关闭时的重要参数解析

其实，在关闭数据库时，经常使用的是 SHUTDOWN 指令，选择不同的参数可满足不同的关闭数据库的要求。

在关闭数据库的步骤中，SHUTDOWN 数据库时需要进行一些额外的操作，如断开连接、回滚数据等，而这些操作依赖于 SHUTDOWN 参数的选择。它有 4 个参数，即：NORMAL、IMMEDIATE、TRANSACTIONAL 和 ABORT。下面分别介绍它们的使用方法。

1. SHUTDOWN NORMAL

这种方式是 SHUTDOWN 数据库的默认方式，如果用户输入 SHUTDOWN，则默认采用 NORMAL 参数，利用这种方式关闭数据库时，不允许新的数据库连接，只有当前的所有连接都退出时才会关闭数据库，这是一种安全地关闭数据库的方式，但是如果有大量用户连接，则需要较长时间关闭数据库。

2. SHUTDOWN IMMEDIATE

这种方式可以较快且安全地关闭数据库，是 DBA 经常采用的一种关闭数据库的方式，此时 Oracle 会做一些操作，如中断当前事务、回滚未提交的事务、强制断开所有用户连接、执行检查点把脏数据写到数据文件中等。虽然参数 IMMEDIATE 有立即关闭数据库的含义，但是它只是相对的概念，如果当前事务很多，且业务量很大，则中断事务以及回滚数据、断开连接的用户等操作都需要时间。

当关闭数据库时，笔者推荐使用 SHUTDOWN IMMEDIATE 指令。

3. SHUTDOWN TRANSACTIONAL

使用 TRANSACTIONAL 参数时，数据库当前的连接继续执行，但不允许新的连接，一旦当前的所有事务执行完毕，则关闭数据库。

显然这种方式在通常情况下，不会快速关闭数据库，因为如果当前的某些事务一直执行，或许会用几天时间关闭数据库。

4. SHUTDOWN ABORT

这是一种很不安全地关闭数据库的方法，最好不要使用该方式关闭数据库。SHUTDOWN ABORT 关闭数据库时，Oracle 会断开当前的所有用户连接，拒绝新的连接，断开当前的所有执行事务，立即关闭数据库。使用这种方式关闭数据库后，当数据库重启时需要进行数据库恢复，因为它不会对未完成事务回滚，也不会执行检查点操作。

12.3　本章小结

本章主要讲解了数据库的启动和关闭过程，以及在启动和关闭过程中涉及的各种文件操作。在启动数据库时，涉及三种状态，即：NOMOUNT、MOUNT 和 OPEN，通常使用 STARTUP 指令选择不同的参数就可以启动数据库到不同的状态来满足不同的业务需要。关闭数据库和启动数据库正好相反，关闭数据库也涉及三个过程，即：CLOSE、DISMOUNT 和 SHUTDOWN。用户可以选择使用不同的参数，这些参数是：NORMAL、IMMEDIATE、TRANSACTIONAL 和 ABORT。在关闭数据库时，最好使用 SHUTDOWN IMMEDIATE 方式，这种方式安全且相对较快，不到万不得已不要使用 SHUTDOWN ABORT 方式，因为这种方式会造成数据丢失，恢复数据库也需要较长时间。读者需要理解数据库启动和关闭时涉及的几个状态和相关的文件操作，这样在数据库启动、关闭的过程中，如果出现问题就可以较好地处理故障点。

第 13 章

控制文件

在数据库的启动过程中需要打开控制文件，控制文件中保存了 Oracle 系统需要的其他文件的存储目录和与物理数据库相关的状态信息。Oracle 系统利用控制文件打开数据库文件、重做日志文件等从而最终打开数据库。本章着重讲解控制文件在数据库启动过程中的作用，以及如何维护控制文件，实现控制文件的多重控制、添加控制文件以及备份和恢复控制文件，本章的内容对于 Oracle 9i、10g 以及 11g 的版本都适用，或许文件目录有所差别，但是维护和管理控制文件的方法是一致的。

13.1 控制文件概述

对于 DBA 来讲，Oracle 数据库控制文件是相当重要的，它是一个非常小的二进制文件，其中记录了数据库的状态信息，如重做日志文件与数据文件的名字和位置、归档重做日志的历史等，它的大小不会超过 64M，但是归档日志的历史记录会让该文件逐渐变大。

控制文件在数据库启动的 MOUNT 阶段被读取，一个控制文件只能与一个数据库相关联，即控制文件和数据库是一对一的关系，因为控制文件的重要性，所以需要将控制文件放在不同磁盘上，以防止控制文件的失效造成数据库无法启动，控制文件的大小在 CREATE DATABASE 语句时被初始化。

数据库启动与控制文件的关系如图 13-1 所示。

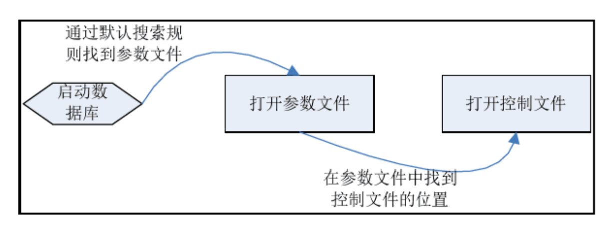


图 13-1 数据库启动和控制文件的关系

上图说明了数据库启动和控制文件的关系，也说明了数据库启动时读取文件的顺序，在数据库启动时，会首先使用默认的规则找到并打开参数文件，在参数文件中保存了控制文件的位置信息（当然还有内存分配等信息），通过参数文件 Oracle 可以找到控制文件的位置，打开控制文件，然后通过控制文件中记录的各种数据库文件的位置打开数据库，从而启动数据库到可用状态。

当成功启动数据库后，在数据库的运行过程中，数据库服务器可以不断地修改控制文件中的内容，所以在数据库被打开的阶段，控制文件必须是可读写的。但是其他任何用户都无法修改控制文件，只有数据库服务器可以修改控制文件中的信息。

1. 查看控件文件的位置

控制文件是数据库启动时非常重要的一个文件，通常一个数据库需要至少 3 个控制文件，而且最好这些控制文件不要放在同一个磁盘上，这样可以防止磁盘故障造成数据库无法启动，那么在 Oracle 数据库上，控制文件作为物理文件到底放在什么地方呢?

在数据库启动和控制文件关系中，控制文件的位置通过参数文件获得，显然我们可以打开参数文件获得控制文件的位置。但是这种方式不方便而且不安全，如果用户不小心输入了某个字符，会造成控制文件错误。Oracle 提供了视图 v$parameter 来查看控制文件的位置，如实例 13-1 所示。

【实例 13-1】使用视图 v$parameter 来查看控制文件的位置。

```
SQL> SELECT value
  2  FROM v$parameter
  3  where name = 'control files';

VALUE
--------------------------------------------------------------------------
C:\oracle\oradata\Lin\CONTROL01.CTL, C:\oracle\oradata\Lin\CONTROL02.CTL,
C:\ora
cle\oradata\Lin\CONTROL03.CTL
```

从输出可以看出，该数据库有三个控制文件，分别为 CONTROL01.CTL、CONTROL02.CTL、CONTROL03.CTL，它们位于同一个目录 C:\oracle\oradata\Lin 下。因为读者的数据库安装目录不同，显示的输出结果或许与上述的输出略有不同，关键是读者要知道数据字典 v$parameter 的作用。也可以使用如下方式查看当前控制文件的位置，如实例 13-2 所示。

【实例 13-2】使用 show parameter 查看当前控制文件的位置。

```
SQL> show parameter control_files;

NAME                           TYPE    VALUE
------------------------------------------------ ----------- ---------
control_files                        string    F:\APP\ADMINISTRATOR\ORADATA\O
                                       RCL\CONTROL01.CTL, F:\APP\ADMI
NISTRATOR\ORADATA\ORCL\CONTROL
                                       02.CTL, F:\APP\ADMINISTRATOR\O
                                       RADATA\ORCL\CONTROL03.CTL
```

使用该方式查看控制文件的位置时，默认输出三列，分别是参数名 NAME、参数类型 TYPE 和参数值 VALUE。显然参数值和实例 13-1 的输出结果一样。

通过数据字典 v$controlfile 也可以查看控制文件的名字和存储目录，如实例 13-3 所示。

【实例 13-3】通过数据字典 v$controlfile 查看控制文件的名字和存储目录。

```
SQL> col name for a50
```

```
SQL> select status , name
  2* from v$controlfile

STATUS  NAME
------- ----------------------------------------------------
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL
```

在 v$parameter 视图中记录了控制文件名字和目录的列名为 VALUE，而在 v$controlfile 视图中记录了控制文件名字和目录的列名为 NAME。从上述三个示例输出可以看出它们的输出结果相同。

2. 查看控制文件的内容

控制文件是二进制文件，是无法通过文本编辑器查看的，而且该文件由 Oracle 数据库服务器自动维护，DBA 无法干预。我们可以通过 Oracle 的文档得知控制文件中的内容，以及使用 v$controlfile_record_section 视图查看所有的记录信息。

控制文件内存放了如下的信息。

- 数据库名：在初始化参数 DB_NAME 中获得，或是在 CREATE DATABASE 语句执行时使用的名字。
- 数据库标识符：数据库创建时 Oracle 记录的标识符。
- 数据库创建时间：创建数据库时由 Oracle 自动记录。
- 表空间信息：当表增加或删除表空间时记录该信息。
- 重做日志文件历史：在日志切换时记录。
- 归档日志文件的位置和状态信息：在归档进程发生时完成。
- 备份的状态信息和位置：由恢复管理器记录。
- 当前日志序列号：日志切换时记录。
- 检验点信息：当检验点事件发生时记录。

控制文件中到底存储了什么内容以及如何查看呢？我们通过实例 13-4 查询控制文件中所存储的内容，此时使用动态数据字典视图 v$controlfile_record_section 来实现。

【实例 13-4】查询控制文件中所存储的内容。

```
SQL> SELECT type,record_size,records_total,records_used
  2  FROM v$controlfile_record_section;

TYPE                 RECORD SIZE RECORDS TOTAL RECORDS USED
---------------------------- -------------------- ---
DATABASE                    192                1              1
CKPT PROGRESS               2036                4              0
REDO THREAD                 104                1              1
REDO LOG                     72                5              3
DATAFILE                    180              100             10
```

```
FILENAME                    524            116              13
TABLESPACE                   68            100              11
TEMPORARY FILENAME           56            100               0
RMAN CONFIGURATION         1108             50               0
LOG HISTORY                  36            226              23
OFFLINE RANGE                56            145               0

TYPE             RECORD_SIZE RECORDS_TOTAL RECORDS_USED
------------------ ----------- ------------- ------------
ARCHIVED LOG                584            230              22
BACKUP SET                   40            101               0
BACKUP PIECE                736            204               0
BACKUP DATAFILE             116            210               0
BACKUP REDOLOG               76             53               0
DATAFILE COPY               660            203               0
BACKUP CORRUPTION            44            185               0
COPY CORRUPTION              40            101               0
DELETED OBJECT               20            407               0
PROXY COPY                  852            210               0
RESERVED4                     1           4072               0

已选择 22 行。
```

从上述输出可以看出控制文件中存放了创建数据库的信息、重做日志信息、数据文件以及归档日志文件记录等信息。

控制文件中记录了大量很有价值的信息用于数据库维护和管理，很多动态数据字典视图就是从控制文件中获得数据的，这些数据字典视图如下所示。

- v$backup。
- v$database。
- v$tempfile。
- v$tablespace。
- v$archive。
- v$log。
- v$logfile。
- v$loghist。
- v$archived_log。
- v$database。

从控制文件中获得相关数据的视图，如 v$database 视图就是从控制文件中获得基础数据，通过该视图可以查看数据库 ID、创建时间和数据库是否处于归档模式等，如下所示：

```
SQL> col name for a20
SQL>select name,created,log_mode
  2* from v$database
```

```
NAME                    CREATED         LOG_MODE
--------------------------------- ------------------------ -
ORCL                     13-10月-09      ARCHIVELOG
```

虽然我们不能从控制文件中直接读取关于数据库创建的信息，但是可以间接地通过数据字典视图 v$database 来查看。上例的输出显示了当前数据库的全局名为 ORCL，创建时间为 13-10 月-09 且处于归档模式。

13.2 存储多重控制文件

正是由于控制文件（Control File）的重要性，就要求控制文件不能只有一个，通常生产数据库中控制文件要多于三个，并且存放在不同的磁盘上。Oracle 数据库会同时维护多个完全相同的控制文件，这也称为多重控制文件。在不同磁盘上存储多重控制文件可以避免控制文件的单点失效问题。如果一个磁盘的控制文件失效，Oracle 会自动使用参数文件中记录的其他控制文件启动数据库。

在控制文件的维护中，Oracle 会建议用户遵循一个原则，即使用多重控制文件，并将控制文件的副本存储在不同的磁盘上，监控备份工作。

在 Oracle 数据库中，控制文件的默认存储目录、数据库文件、重做日志文件等存放在同一个目录下，以 Oracle 11g 为例，该目录为： $ORACLE_BASE\oradata\ORACLE_SID。在笔者安装的 Oracle 11g 中，ORACLE_BASE 为 F:\APP\ADMINISTRATOR，而 ORACLE_SID 为 ORCL，所以控制文件目录为：F:\APP\ADMINISTRATOR\ORADATA\ORCL。Oracle 会同时建立三个控制文件，默认控制文件名依次为 CONTROL1.CTL、CONTROL2.CTL、CONTROL3.CTL，如实例 13-5 所示。

【实例 13-5】查看当前数据库上的控制文件分布。

```
SQL> col name for a50
SQL> select status,name
  2  from v$controlfile;

STATUS  NAME
------------- -----------------------------------------------------
       F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL
       F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL
       F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL
```

把这么重要的控制文件放在同一个磁盘的同一个目录下显然是不安全的，我们应该遵循 Oracle 的忠告，使用多重控制文件并存储在不同的磁盘上。

数据库在启动时首先要读取参数文件，而参数文件有传统的 PFILE（init.ora）文件和 SPFILE 文件，针对采用不同的数据库启动初始化参数文件来实现控制文件的分布式存储的方式也会略有不同。下面将依次演示。

13.2.1　移动控制文件

1. 使用 PFILE 文件时移动控制文件

PFILE 文件是一个可识别的正文文件，我们可以对存储在 PFILE 中的参数直接更改，这就方便了使用 PFILE 实现移动控制文件的存储方式，其具体步骤如下。

（1）利用数据字典获得控制文件的名字

【实例 13-6】利用数据字典 v$parameter 获得控制文件的名字。

```
SQL> select value
  2  from v$parameter
  3  where name = 'control_files';

VALUE
--------------------------------------------------------------------------
F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL,F:\APP\ADMINISTRATOR\
ORADATA\ORCL\CONTROL02.CTL,F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL
```

此时输出的控制文件信息就是参数文件中保存的控制文件名及目录信息，我们也可以打开参数文件来验证，如图 13-2 所示。

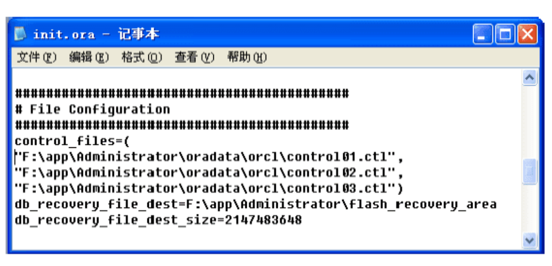


图 13-2　参数文件中的控制文件信息

（2）关闭数据库

【实例 13-7】关闭数据库。

```
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
```

（3）修改参数值

修改参数文件 PFILE 中参数 control_files 的值，即更改控制文件名而使得该文件存储在不同目录下，并保存该文件。修改结果如图 13-3 所示。

修改之后，保存该文件。使用操作系统命令把控制文件 CONTROL2.CTL 和 CONTROL2.CTL 分别拷贝到目录 D:\OraBackup\disk1 和 D:\OraBackup\disk2 下。之后需要删除默认目录下的 CONTROL2.CTL 和 CONTROL3.CTL 文件，以防止文件冗余。

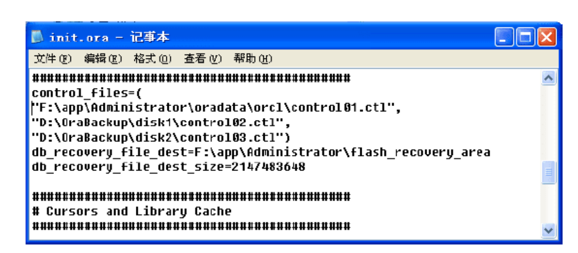


图 13-3 修改参数文件中的控制文件名

（5）重启数据库

【实例 13-8】重启数据库。

```
SQL> startup
ORACLE 例程已经启动。

Total System Global Area  118255568 bytes
Fixed Size                  282576 bytes
Variable Size             83886080 bytes
Database Buffers           33554432 bytes
Redo Buffers                532480 bytes
数据库装载完毕。
数据库已经打开。
```

（6）验证修改结果

【实例 13-9】验证控制文件的修改结果。

```
SQL> col name for a40
SQL>SELECT *
  2* FROM v$controlfile

STATUS  NAME
------- ----------------------------------------
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL
        D:\ORABACKUP\DISK1\CONTROL02.CTL
        D:\ORABACKUP\DISK2\CONTROL03.CTL
```

此时，我们的修改成功，三个控制文件存储在不同磁盘上（CONTROL02.CTL 和 CONTROL03.CTL 读者可以理解为存储在不同的磁盘，因为笔者的计算机只有两个磁盘分区，所以在 D 盘的目录 ORABACKUP 下使用 disk1 和 disk2 模拟不同的磁盘），三个不同磁盘上的控制文件由 Oracle 数据库服务器自动维护，一旦发生诸如增删数据文件或更改重做日志名等事件时，Oracle 数据库服务器会同时更改这三个控制文件中的信息。

2. 使用 SPFILE 文件时移动控制文件

因为 SPFILE 是二进制文件，所以无法通过修改 PFILE 的方式修改控制文件名，Oracle 提供了 ALTER SYSTEM 指令允许修改 SPFILE 中的参数。此时实现控制文件分布式存储的步骤如下。

（1）获取控制文件名

如果读者清楚自己的控制文件信息，此步骤可以省略。

【实例 13-10】获取控制文件名。

```
SQL> select *
  2  from v$controlfile;

STATUS  NAME
-------- ----------------------------------------
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL
```

（2）使用 alter system set 指令修改 SPFILE 中的控制文件名

【实例 13-11】使用 alter system set 指令修改 SPFILE 中的控制文件名。

```
SQL> alter system set control_files =
  2  ' F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL ',
  3  'D:\ORABACKUP\DISK3\CONTROL02.CTL',
  4  'D:\ORABACKUP\DISK4\CONTROL03.CTL' SCOPE = SPFILE;

系统已更改。
```

（3）关闭数据库

【实例 13-12】关闭数据库。

```
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
```

（4）复制文件

将控制文件 CONTROL02.CTL 和 CONTROL03.CTL 复制到更改的目录下，即将 CONTROL02.CTL 复制到目录 D:\ORABACKUP\DISK3 下，将 CONTROL03.CTL 复制到目录 D:\ORABACKUP\DISK4 下。

（5）重启数据库

【实例 13-13】正常启动数据库。

```
SQL> conn /as sysdba
已连接到空闲例程。
SQL> startup
ORACLE 例程已经启动。

Total System Global Area  539856896 bytes
Fixed Size                  1334380 bytes
Variable Size             247464852 bytes
Database Buffers          281018368 bytes
Redo Buffers               10039296 bytes
数据库装载完毕。
```

```
数据库已经打开。
```

我们再验证一下是否使用 SPFILE 启动了数据库，如下所示。

【实例 13-14】验证是否使用 SPFILE 启动了数据库。

```
SQL> show parameter spfile;

NAME                    TYPE       VALUE
------------------------------------ -----------
spfile                      string      F:\APP\ADMINISTRATOR\PRODUCT\1
                                       1.1.0\DB_1\DATABASE\SPFILEORCL
                                       .ORA
```

显然，现在 VALUE 的值不为空，说明此时已使用 SPFILE 文件启动了数据库。

（6）验证修改结果

【实例 13-15】验证控制文件的修改结果。

```
SQL> select status,name
  2  from v$controlfile;

STATUS  NAME
------- ----------------------------------------
        F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL
        D:\ORABACKUP\DISK3\CONTROL02.CTL
        D:\ORABACKUP\DISK4\CONTROL03.CTL
```

从上述输出可以看出，我们已经将名为 CONTROL02 和 CONTROL03 的控制文件分布到了 D:\ORABACKUP\DISK3 和 D:\ORABACKUP\DISK4 目录下。

13.2.2 添加控制文件

Oracle 默认建立三个控制文件，使用上节中介绍的方法可以移动控制文件，将控制文件存储在不同的磁盘空间，以防止控制文件的单点失效。在生产数据库中往往至少需要三个控制文件，如果我们需要 5 个控制文件该如何处理呢？下面给出一个具体步骤，但不作详细过程演示，以数据库启动时采用 PFILE 参数文件为例。

01 查看控制文件的名字。

02 关闭数据库。

03 使用操作系统命令将步骤 1 中的一个控制文件复制到一个目录下，并修改控制文件的名字，如复制到目录 D:\OraBackup\disk5 下，控制文件名修改为 CONTROL05.CTL。

04 修改参数文件中参数 control_files 的值，添加一个控制文件名，如 D:\OraBackup\disk5\CONTROL05.CTL。

05 重新启动数据库。

13.3　备份和恢复控制文件

由于控制文件在数据库启动过程中非常重要，所有最好备份控制文件，这样在发生控制文件损坏时，使用备份的文件来恢复控制文件，从而保证数据库的正常启动和运行。

13.3.1　备份控制文件

由于控制文件的重要性，备份控制文件是非常必要的，这也是 Oracle 极力推荐的，备份控制文件如实例 13-16 所示。

【实例 13-16】使用 ALTER DATABASE BACKUP CONTROL 备份控制文件。

```
SQL> alter database backup controlfile to
2  'd:\OraBackup\disk5\backup_controlfile_09_06_14.ora';

数据库已更改。
```



目录 d:\OraBackup\disk5 必须是存在的，Oracle 会自动在该目录下创建一个备份文件 backup_controlfile_09_06_14.ora。由于 Oracle 数据库服务器不断地更改控制文件中的信息，所以备份的控制文件不是最新的，在恢复数据库时最好不要使用备份的控制文件，这样会造成数据丢失。

下面通过 Windows 的资源管理器来查看备份的文件是否存在，如图 13-4 所示。

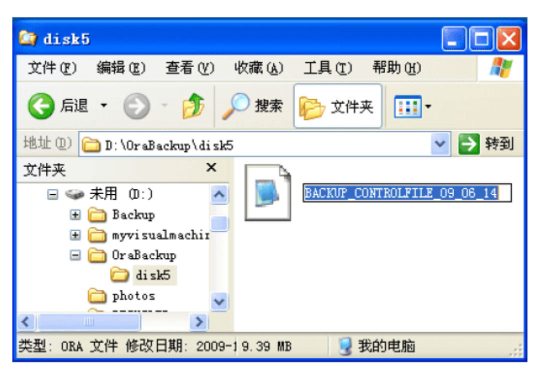


图 13-4　查看备份的控制文件

Oracle 也提供了一种方式用于备份控制文件，即将控制文件备份到追踪文件中，使用该文件就可以重建控制文件。其备份方法首先要设置参数 sql_trace 为 true，如实例 13-17 所示。

【实例 13-17】设置参数 sql_trace 为 true。

```
SQL> alter session set sql_trace = true;

会话已更改。
```



在使用该方式备份控制文件时，必须把参数 sql_trace 的值设置为 true，Oracle 默认该参数值为 false。再使用如下指令将控制文件备份到追踪文件中。

```
SQL> alter database backup controlfile to trace;

数据库已更改。
```

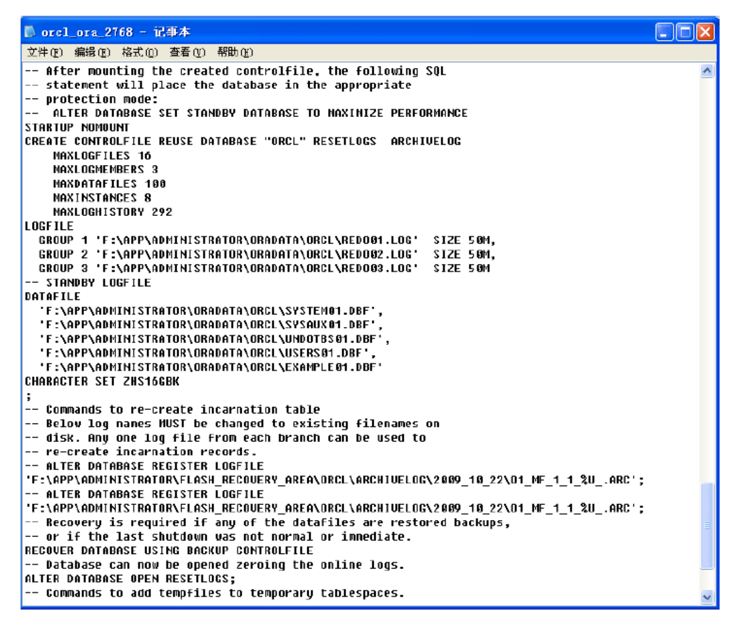
Oracle 提供了一个参数 user_dump_dest，可以查看跟踪文件的存储目录。使用实例 13-18 查询该存储目录。

【实例 13-18】查询参数 user_dump_dest 指定的目录。

```
SQL> show parameter user_dump_dest;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ----
user_dump_dest                       string      f:\app\administrator\diag\rdbm
                                                 s\orcl\orcl\trace
```

从输出可以看出，跟踪文件的存储目录为 f:\app\administrator\diag\rdbm s\orcl\orcl\trace，打开该目录，按照时间顺序对该目录下的文件进行排序，打开最近建立的跟踪文件，如图 13-5 所示。




```
-- After mounting the created controlfile, the following SQL
-- statement will place the database in the appropriate
-- protection mode:
--  ALTER DATABASE SET STANDBY DATABASE TO MAXIMIZE PERFORMANCE
STARTUP NOMOUNT
CREATE CONTROLFILE REUSE DATABASE "ORCL" RESETLOGS  ARCHIVELOG
    MAXLOGFILES 16
    MAXLOGMEMBERS 3
    MAXDATAFILES 100
    MAXINSTANCES 8
    MAXLOGHISTORY 292
LOGFILE
  GROUP 1 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG'  SIZE 50M,
  GROUP 2 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG'  SIZE 50M,
  GROUP 3 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG'  SIZE 50M
-- STANDBY LOGFILE
DATAFILE
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSTEM01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSAUX01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\UNDOTBS01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\USERS01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\EXAMPLE01.DBF'
CHARACTER SET ZHS16GBK
;
-- Commands to re-create incarnation table
-- Below log names MUST be changed to existing filenames on
-- disk. Any one log file from each branch can be used to
-- re-create incarnation records.
-- ALTER DATABASE REGISTER LOGFILE
'F:\APP\ADMINISTRATOR\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2009_10_22\O1_MF_1_1_%U_.ARC';
-- ALTER DATABASE REGISTER LOGFILE
'F:\APP\ADMINISTRATOR\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2009_10_22\O1_MF_1_1_%U_.ARC';
-- Recovery is required if any of the datafiles are restored backups,
-- or if the last shutdown was not normal or immediate.
RECOVER DATABASE USING BACKUP CONTROLFILE
-- Database can now be opened zeroing the online logs.
ALTER DATABASE OPEN RESETLOGS;
-- Commands to add tempfiles to temporary tablespaces.
```

图 13-5　建立控制文件的跟踪文件内容

13.3.2　恢复控制文件

在数据库中如果有一个或多个控制文件丢失或出错，就可以根据不同的情况进行处理。

1. 部分控制文件损坏的情况

如果数据库正在运行，我们应先关闭数据库，再将完好的控制文件复制到已经丢失或出错的控制文件的位置，但是要更改该丢失或出错的控制文件的名字。如果存储丢失的控制文件的目录也被破坏，则需要重新建立一个新的目录，用于存放新的控制文件，并为该控制文件命名。此时需要修改数据库初始化参数文件中控制文件的位置信息。

2. 控制文件全部丢失或损坏的情况

此时使用备份的控制文件（这也是为什么 Oracle 强调在数据库结构发生变化后要进行控制文件备份的原因）重建控制文件。先关闭数据库，再将备份的控制文件拷贝到先前控制文件的所在位置上，并更改备份控制文件名为先前控制文件的文件名。

接下来打开数据库到 MOUNT 状态，代码如下所示：

```
SQL>startup mount
```

然后打开数据库，代码如下所示：

```
ALTER DATABASE OPEN USING BACKUP CONTROLFILE;
```



此时由于使用备份的控制文件，所以会有数据丢失的情况，因为从该备份文件被备份时刻起到控制文件发生故障的时间段内发生的数据变化无法恢复。

3. 通过跟踪文件重建控制文件

跟踪文件中记录了用于建立控制文件的 SQL 语句，适当的编辑跟踪文件，然后使用 SQL 指令重建控制文件，如从 STARTUP NOMOUNT 到 CHARACTER SET ZHS16GBK 的部分提取出来，再增加一些指令来制作脚本文件 createctl.sql，将下面的指令保存为 createctl.sql 脚本文件：

```
STARTUP NOMOUNT
CREATE CONTROLFILE REUSE DATABASE "ORCL" RESETLOGS  ARCHIVELOG
    MAXLOGFILES 16
    MAXLOGMEMBERS 3
    MAXDATAFILES 100
    MAXINSTANCES 8
    MAXLOGHISTORY 292
LOGFILE
  GROUP 1 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG'  SIZE 50M,
  GROUP 2 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG'  SIZE 50M,
  GROUP 3 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG'  SIZE 50M
-- STANDBY LOGFILE
DATAFILE
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSTEM01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSAUX01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\UNDOTBS01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\USERS01.DBF',
  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\EXAMPLE01.DBF'
CHARACTER SET ZHS16GBK
```

```
;
RECOVER DATABASE
ALTER SYSTEM ARCHIVE LOG ALL;
ALTER DATABASE OPEN;
ALTER TABLESPACE TEMP ADD TEMPFILE ' C:\ORACLE\ORADATA\LIN \TEMP01.DBF'
    SIZE 41943040  REUSE AUTOEXTEND ON NEXT 655360  MAXSIZE 32767M;
```

要运行该脚本文件：

```
SQL> @createctl.sql
SQL> STARTUP NOMOUNT
ORACLE instance started.

Total System Global Area  135337420 bytes
Fixed Size                  452044 bytes
Variable Size             109051904 bytes
Database Buffers           25165824 bytes
Redo Buffers                667648 bytes
SQL> CREATE CONTROLFILE REUSE DATABASE "   ORCL" NORESETLOGS  ARCHIVELOG
  2  --  SET STANDBY TO MAXIMIZE PERFORMANCE
  3      MAXLOGFILES16
  4      MAXLOGMEMBERS 3
  5      MAXDATAFILES 100
  6      MAXINSTANCES8
  7      MAXLOGHISTORY 92
  8  LOGFILE
  9   GROUP 1 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG'  SIZE 50M,
 10   GROUP 2 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG'  SIZE 50M,
 11   GROUP 3 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG'  SIZE 50M,
 12   DATAFILE
 13 'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSTEM01.DBF',
14  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\SYSAUX01.DBF',
15  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\UNDOTBS01.DBF',
16  'F:\APP\ADMINISTRATOR\ORADATA\ORCL\USERS01.DBF',
17   'F:\APP\ADMINISTRATOR\ORADATA\ORCL\EXAMPLE01.DBF'
18    CHARACTER SET ZHS16GBK
;

Control file created.

SQL> RECOVER DATABASE
ORA-00283: recovery session canceled due to errors
ORA-00264: no recovery required
SQL> ALTER SYSTEM ARCHIVE LOG ALL;

System altered.

SQL> ALTER DATABASE OPEN;

Database altered.
```

```
SQL>      ALTER      TABLESPACE      TEMP      ADD      TEMPFILE
'F:\APP\ADMINISTRATOR\ORADATA\ORCL\TEMP01.DBF'
     SIZE 28311552  REUSE AUTOEXTEND ON NEXT 655360  MAXSIZE 32767M;

Tablespace altered.

SQL>
```

在使用 TRACE 文件重新建立数据库控制文件时，通过 TRACE 文件知道该数据库的日志文件、数据文件、数据库名和其他一些参数信息，通过执行一个编辑好的脚本文件可以重新建立一个可用的控制文件。在用户的具体操作中需要依据数据库服务器本身的具体情况具体分析。

4. 手工重建控制文件

在使用 TRACE 文件恢复数据库控制文件的过程中，读者已经看到了使用一个编辑好的脚本文件创建控制文件的过程，如果在脚本文件中的参数都很清楚，当然可以使用 CREATE CONTROLFILE 指令逐步创建控制文件。

13.4 本章小结

控制文件在数据库的启动过程中起着关键作用，当数据库启动时，需要通过控制文件找到数据文件、日志文件的位置，同时需要检查控制文件中记录的检查点 SCN 序列号和数据文件中的检查点序列号对比来实现数据库实例的恢复，如果数据库控制文件损坏，则数据库无法启动。由于控制文件的重要性，Oracle 支持存储多重控制文件，并且最好把控制文件存储在不同的磁盘目录下，读者在理解了多重控制文件的概念后，进一步掌握移动控制文件和添加控制文件来实现控制文件的维护，数据库控制文件的备份和控制是 DBA 需要掌握的内容。

第 14 章

参 数 文 件

本章将主要讲解 Oracle 数据库实例维护和相关的参数文件，参数文件是数据库实例启动时需要使用的文件，初始化参数文件用于控制数据库的行为，如数据库的资源限制、进程数限制、各种内存的分配空间、控制文件的位置等，其中大部分参数使用系统的默认值。在数据库实例启动时，会读取初始化参数文件、读取各种参数的值来初始化数据库。如果没有初始化参数文件，数据库则无法启动，所以如果发生参数文件损坏，则需要恢复，在本章也会详细介绍如何恢复损坏的控制文件。

14.1 参数文件概述

在 Oracle 数据库体系结构中，参数文件是很重要的一个文件，在数据库实例（Instance）启动时，Oracle 会读取该文件中的参数来为实例分配内存、获得一些资源的位置、设置用户进程、获得控制文件的位置以及用户登录的信息。在初始化参数文件中，显示了给出的参数值，而大多数的参数采用系统的默认值。

1. 什么是参数文件

在 Oracle 数据库中有两类初始化参数文件，即在 Oracle 8g 之后的 PFILE 文件和 Oracle 9i 之后（Oracle 10g、Oracle 11g）的 SPFILE 文件。其中 PFILE 文件是一个静态的正文文件，可以使用文本编辑器编辑。而 SPFILE 是动态的二进制文件，只能通过 Oracle 的指令修改。下面解释一下静态和动态的含义。

- 静态：文件修改后不会在当前实例中生效，只有重新启动实例后，所做的修改才会生效。
- 动态：文件修改后直接在实例中生效。

我们通常也把数据库的参数文件称为数据库初始化文件，如把 PFILE 和 SPFILE 都称为数据库初始化文件，PFILE 文件名默认为 init<ORACLE_SID>.ora，SPFILE 的文件名默认为 spfile<ORACLE_SID>.ora。

为了更直观地理解初始化参数的内容，我们给出该 SPFILE 文件的部分内容，该文件为二进制文件，无法直接查看，我们需要一个创建 PFILE 的操作，转换成正文文件，如实例 14-1 所示。

【实例 14-1】从 SPFILE 创建 PFILE 的操作。

```
SQL> create pfile = 'd:/init_2009_06_11.ora' from spfile;

文件已创建。
```



此时创建了一个 PFILE 文件，PFILE 文件是正文文件，它是可读的.ora 文件，使用文本工具可以打开该文件。

然后，用文本编辑器打开文件 d:/init_2009_06_11.ora，如图 14-1 所示。

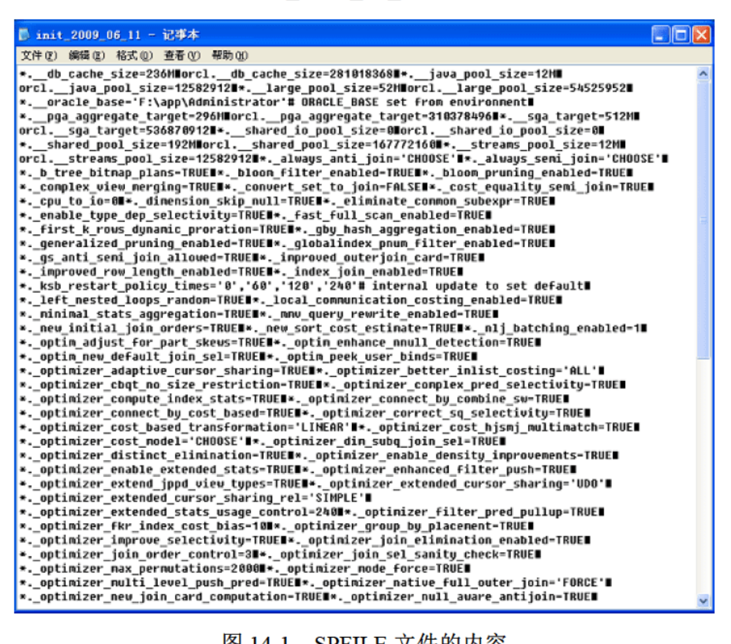


图 14-1　SPFILE 文件的内容

从上图可以看出，初始化参数文件的内容如下。

- 告警文件和其他后台追踪文件的存储目录。
- 控制文件的位置和名字。
- 是否开启数据库缓冲顾问。
- 数据块的大小。
- 数据库缓冲区大小。
- 数据库名和实例名。
- Java 池和大池的大小。
- 归档日志文件的存储位置及是否启动归档。
- 实例可以同时启动的进程数。
- 打开的游标数。
- 还原段和还原空间的配置。

- Timed_statistic = ture。
- User_dump_dest = 'c:\oracle\admin\Lin\udump'等。

2. 什么是参数

在简单讲解了初始化参数文件 PFILE（init<ORACLE_SID>.ora）和 SPFILE 之后，我们需要知道什么是参数。

数据库初始化参数由一系列的“参数/参数值”对组成，如图 14-1 所示的 SPFILE 文件中，参数 db_name ='orcl'，此时 db_name 是参数名，而参数值就是 orcl。相信读者已经理解了参数对的含义。

在 Oracle 数据库中，要查看这些参数可以通过动态视图 v$parameter、在 SQL*Plus 中使用 show parameter 命令来查看。实例 14-2 是使用动态视图 v$parameter 查询数据库的初始化参数，在查询之前为了了解视图 v$parameter 中有哪些列，可使用 desc 指令查看。

【实例 14-2】查询视图 v$parameter 的结构。

```
SQL> desc v$parameter
 名称                                                是否为空? 类型
 ----------------------------------------- -------- ----------------
 NUM                                                NUMBER
 NAME                                               VARCHAR2(64)
 TYPE                                               NUMBER
 VALUE                                              VARCHAR2(512)
 ISDEFAULT                                          VARCHAR2(9)
 ISSES_MODIFIABLE                                   VARCHAR2(5)
 ISSYS_MODIFIABLE                                   VARCHAR2(9)
 ISMODIFIED                                         VARCHAR2(10)
 ISADJUSTED                                         VARCHAR2(5)
 DESCRIPTION                                        VARCHAR2(64)
 UPDATE_COMMENT                                     VARCHAR2(255)
```

我们再查看参数 db_name 的值：

```
SQL> select value
  2  from v$parameter
  3  where name = 'db_name';

VALUE
--------------------------------------------------------------------------
ORCL
```

下面使用 show parameter 的方式查询初始化参数文件中的参数值。

【实例 14-3】查询初始化参数文件中 db_name 参数值。

```
SQL> show parameter db_name;

NAME                                 TYPE        VALUE
------------------------------------ ----------- -
db_name                              string      ORCL
```

笔者建议读者使用 show parameter 方式，因为这种方式更简洁，符合思维习惯，而且它会自动完成通配符的功能，如输入 show parameter db 则会自动在 db 前后添加“%”，会显示所有和 db 相关的参数名和参数值，如实例 14-4 所示。

【实例 14-4】查询所有和 db 相关的参数名和参数值。

```
SQL> show parameter db

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------
db_16k_cache_size                    big integer 0
db_2k_cache_size                     big integer 0
db_32k_cache_size                    big integer 0
db_4k_cache_size                     big integer 0
db_8k_cache_size                     big integer 0
db_block_buffers                     integer     0
db_block_checking                    string      FALSE
db_block_checksum                    string      TYPICAL
db_block_size                        integer     8192
db_cache_advice                      string      ON
db_cache_size                        big integer 0

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
db_create_file_dest                  string
db_create_online_log_dest_1          string
db_create_online_log_dest_2          string
db_create_online_log_dest_3          string
db_create_online_log_dest_4          string
db_create_online_log_dest_5          string
db_domain                            string
db_file_multiblock_read_count        integer     128
db_file_name_convert                 string
db_files                             integer     200
db_flashback_retention_target        integer     1440

NAME                                 TYPE        VALUE
------------------------------------ ----------- ----
db_keep_cache_size                   big integer 0
db_lost_write_protect                string      NONE
db_name                              string      orcl
db_recovery_file_dest                string      F:\app\Administrator\flash
                                                 overy_area
db_recovery_file_dest_size           big integer 2G
db_recycle_cache_size                big integer 0
db_securefile                        string      PERMITTED
db_ultra_safe                        string      OFF
db_unique_name                       string      orcl
db_writer_processes                  integer     1
```

```
NAME                                 TYPE        VALUE
------------------------------------ -----------
dbwr_io_slaves                        integer     0
rdbms_server_dn                       string
standby_archive_dest                    string      %ORACLE_HOME%\RDBMS
standby_file_management                string      MANUAL
xml_db_events                         string      enable
```

14.2 静态参数文件

静态参数文件是指在 Oracle 8g 之前使用的初始化参数文件，该文件的默认文件名为 init.ora，在笔者的 Oracle 11g 数据库中，初始化静态参数文件名为 init.ora，在 Windows 平台默认的存储目录为 $ORACLE_BASE\admin\orcl\pfile，而 v$ORACLE_BASE 为 F:\app\Administrator，所以 Oracle 11g 数据库中的静态参数文件的默认存储目录为 F:\app\Administrator\admin\orcl\pfile。如图 14-2 所示为 Windows 平台上静态初始化参数的存储目录。

静态初始化参数文件是可读的参数文件，可以使用“记事本”工具打开进行查看或修改参数文件中的参数，如图 14-3 所示。

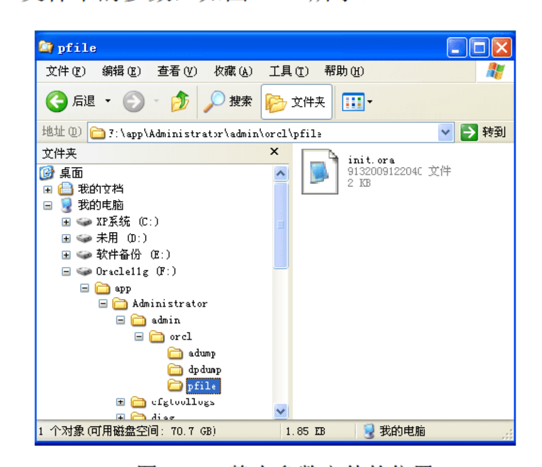


图 14-2 静态参数文件的位置

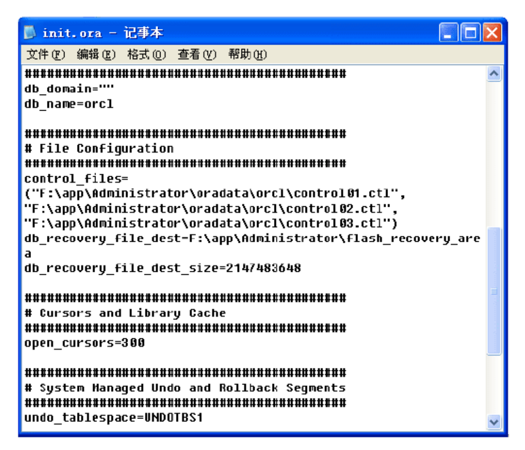


图 14-3 静态初始化参数文件的内容

在数据库启动时，会自动搜索需要的初始化参数文件，但是在启动时可以指定使用 PFILE 启动数据库，如果读者想尝试修改某些参数，又想知道修改后的效果可以采用这种方式。下面利用在实例 14-1 中创建的静态参数文件来启动数据库，实例 14-5 说明了如何使用 PFILE 文件启动数据库。

【实例 14-5】使用 PFILE 文件启动数据库。

```
SQL> startup pfile = 'd:\init_2009_06_11.ora';
ORACLE 例程已经启动。

Total System Global Area  539856896 bytes
Fixed Size                  1334380 bytes
```

```
Variable Size            247464852 bytes
Database Buffers         281018368 bytes
Redo Buffers             10039296 bytes
数据库装载完毕。
数据库已经打开。
```



使用 PFILE 文件启动数据库时，如果使用 alter system 修改参数后，该参数不会在 PFILE 文件中修改，要保持修改一致，则需要手动修改 PFILE 文件中的相应参数。这或许是 Oracle 引入 SPFILE 的原因吧！

通过刚才的分析可知，上述方式启动数据库实例时，Oracle 首先找到 PFILE 指定的初始化文件，在笔者的计算机上为 d:\init_2009_06_11.ora 文件。其实，可以将文件 $ORACLE_BASE\admin\orcl\pfile \init.ora 的内容直接拷贝到另一个存储目录下，如目录 e:\backup_init.ora 中，这样就实现了静态参数文件的备份。使用操作系统指令完成拷贝操作，如实例 14-6 所示。

【实例 14-6】拷贝文件。

```
C:\>copy d:\init_2009_06_11.ora e:\backup_init.ora
已复制         1 个文件。
```

此时，查看 e:\backup_init.ora 文件的内容，会发现在 e:\下创建了一个文件 backup_init.ora，该文件是静态参数文件的一个备份，可以查看该文件的内容，如图 14-4 所示。

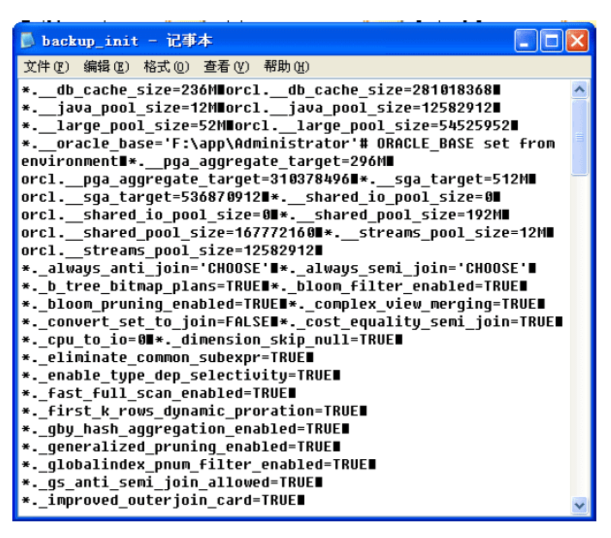


图 14-4　backup_init.ora 文件内容

在 Oracle 11g 中虽然默认启动数据库时使用 SPFILE，但是仍然保留了 PFILE 参数文件。建议读者使用数据库的默认启动方式，即采用动态参数文件启动数据库，这样就可以动态地修改一些运行数据库参数，对于不允许中断服务的生产数据库而言尤为重要。

14.3　服务器动态参数文件

服务器动态参数文件，即 SPFILE 文件，该文件是一个二进制文件，存储在数据库服务器上，要想操作或修改该文件只能在服务器上操作。二进制文件无法使用文本编辑器修改，但是可以使用 alter system 指令修改参数值。该文件的默认文件名基于不同的操作系统平台而不同，在 Windows 平台下为 spfile<ORACLE_SID>.ora，在 UNIX 平台下为 spfileORACLE_SID.ora。对于 SPFILE 文件的维护，最好保持该文件的默认存储目录，这样不会影响 SPFILE 的简单性。在实际中很少维护该文件，也很少出现 SPFILE 文件损坏的情况。在 Oracle 11g 中默认使用 SPFILE 启动数据库。鉴于市场上依然存在 9i 数据库，下面以 Oracle 9i 版本为基础介绍 SPFILE，在 Oracle 11g 和 10g 中维护 SPFILE 的方法相同，差异仅在于 SPFILE 的存储目录不同，其实关键还是 Oracle 11g 的 $ORACLE_HOME 不同，其 SPFILE 同样存储在$ORACLE_HOME\database 下。

我们知道 SPFILE 的默认存储位置在 Windows 平台下为$ORACLE_HOME\database，在 UNIX 平台下为$ORACLE_HOME/dbs。图 14-5 是笔者计算机上 SPFILE 的位置，其文件名为 SPFILEORCL.ora。

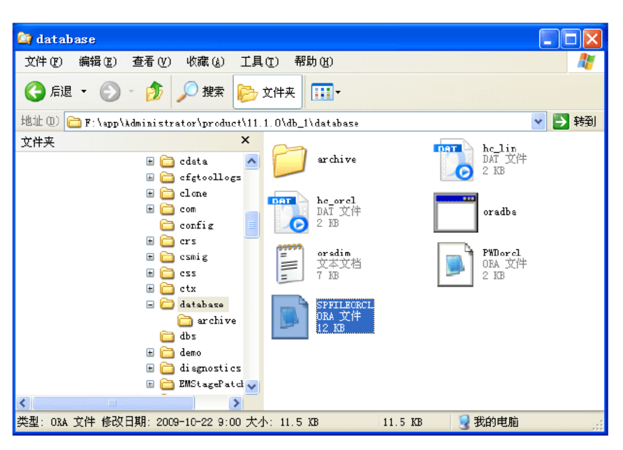


图 14-5　SPFILE 文件的存储目录

14.3.1　创建 SPFILE 文件

动态参数文件 SPFILE 通过静态参数文件 PFILE 创建，其语法格式为：

```
create spfile [= 'spfile文件名'] pfile = [= 'pfile文件名']
```

其中 spfile 文件名为要创建的 SPFILE 的名字，而 PFILE 文件名是存在于服务器上的 PFILE 文件名。

假设一个数据库使用 PFILE 启动数据库，SPFILE 丢失，而 PFILE 文件存储在默认目录下（$ORACLE_HOME\database），则可以使用 create spfile from file 在 SPFILE 的默认目录下创建默认的 SPFILE 文件。如果不使用 PFILE 的默认目录，也可以使用如下指令创建 SPFILE：

```
create spfile from pfile = 'pfile 文件名'
```

下面演示这个步骤。

1. 用 PFILE 启动数据库

使用 PFILE 启动数据库，但不使用默认方式，示例如下：

```
SQL> startup pfile = 'd:\init_2009_06_11.ora';
ORACLE 例程已经启动。

Total System Global Area  539856896 bytes
Fixed Size                  1334380 bytes
Variable Size             247464852 bytes
Database Buffers          281018368 bytes
Redo Buffers               10039296 bytes
数据库装载完毕。
数据库已经打开。
```

2. 验证是否使用 SPFILE 启动数据库

示例如下：

```
SQL> show parameter spfile;

NAME                                 TYPE        VALUE
------------------------------------ -----------
spfile                               string
```

使用 show parameter spfile 查看参数 spfile 的值，如果该值为空，说明没有使用 SPFILE 作为数据库启动的初始化参数。

3. 创建 SPFILE 文件

示例如下：

```
SQL> create spfile from pfile = 'd:\init_2009_06_11.ora'
  2  ;

文件已创建。
```

此时查看 SPFILE 的默认存储目录，可以查看到刚才创建的 SPFILE，不过我们使用默认的文件名，在笔者的计算机上，其目录为$ORACLE_HOME \database，文件名为 SPFILEORCL.ora。下次启动数据库时，如果使用 startup 指令，则默认首先使用 SPFILE 启动数据库。

4. 重启数据库

关闭用 PFILE 启动的数据库，使用 startup 指令重新启动数据库，示例如下：

```
SQL> startup
ORACLE 例程已经启动。

Total System Global Area  118255568 bytes
```

```
Fixed Size                 282576 bytes
Variable Size             83886080 bytes
Database Buffers          33554432 bytes
Redo Buffers               532480 bytes
数据库装载完毕。
数据库已经打开。
```

5. **再次验证数据库**

再次验证数据库启动时选择初始化参数文件：

```
SQL> show parameter spfile

NAME                                 TYPE        VALUE
------------------------------------ ----------- ----
spfile                               string   F:\APP\ADMINISTRATOR\PRODUCT\1
1.1.0\DB_1\DATABASE\SPFILEORCL.ORA
```



笔者使用 d:\init_2009_06_11.ora 下的静态初始化参数来创建 SPFILE，如果使用 create spfile from pfile，则将使用默认的静态初始化参数来创建动态服务器参数文件。

14.3.2 维护 SPFILE 文件

如果 SPFILE 损坏该如何处理呢？当然启动数据库可以使用 PFILE，但这样会造成一些修改后的参数值丢失。我们可以使用 PFILE 恢复 SPFILE。但是更好的方式是采用从告警日志文件中创建 SPFILE，因为告警日志文件中记录了数据库启动时的参数和参数值。其思路是把告警日志文件中部分参数信息拷贝进 SPFILE 文件中，再使用 create spfile from pfile 指令创建 spfile。告警日志文件中记录的部分参数文件信息如下：

```
System parameters with non-default values:
  processes                = 100
  timed_statistics          = TRUE
  shared pool size          = 29360128
  large_pool_size           = 1048576
  java_pool_size            = 33554432
  control_files             = C:\oracle\oradata\Lin\CONTROL01.CTL,
C:\oracle\oradata\Lin\CONTROL02.CTL, C:\oracle\oradata\Lin\CONTROL03.CTL
  db_block_size             = 4096
  db_cache_size             = 33554432
  db_cache_advice           = ON
  compatible               = 9.0.0
  log_archive_start         = TRUE
  log_archive_dest_1        = LOCATION=D:\oracle\online_dbbackup\arch
  fast start mttr target    = 300
  undo_management           = AUTO
  undo_tablespace           = UNDOTBS
  remote_login_passwordfile= EXCLUSIVE
```

```
  db_domain               =
  instance_name            = Lin
  dispatchers             = (PROTOCOL=TCP)(SER=MODOSE),
(PROTOCOL=TCP)(PRE=oracle.aurora.server.GiopServer),
(PROTOCOL=TCP)(PRE=oracle.aurora.server.SGiopServer)
  background_dump_dest     = C:\oracle\admin\Lin\bdump
  user_dump_dest           = C:\oracle\admin\Lin\udump
  core_dump_dest           = C:\oracle\admin\Lin\cdump
  sort_area_size           = 524288
  db_name                 = Lin
  open_cursors             = 300
```

将上述的参数拷贝到 SPFILE 的 init<ORACLE_SID>.ora 文件，然后使用 create spfile from pfile 指令在 SPFILE 的默认目录下创建一个名为 spfile<ORACLE_SID>.ora 的 SPFILE。

14.3.3　修改 SPFILE 中的参数值

在使用 SPFILE 作为初始化参数文件启动数据库后，根据需要可以动态调整初始化的参数值。要修改 SPFILE 中的文件值，不能使用文本编辑器编辑，因为它是二进制文件。Oracle 提供了 ALTER SYSTEM 命令来修改 SPFILE 中的参数值。其语法格式如下所示：

```
ALTER SYSTEM SET parameter = value <comment = 'text'> <deferred>
<scope = memory|spfile|both> <sid = 'sid|*'>
```

参数说明如下：

- parameter = values：为某个参数 parameter 赋值，如 sort_area_size = 65536，即把参数 sort_area_size 的值设置为 65536 字节。
- <comment = 'text'>：该参数是可选的，提供注释信息，text 为字符串，用于附加一些注释信息，这个注释位于 v$parameter 数据字典视图中的 update_comment 字段中。
- <deferred>：该参数说明修改是否对当前会话有效，默认情况下，参数修改立即生效，但有些参数要求对新的会话生效。可以使用实例 14-7 查看哪些参数需要延迟生效。

【实例 14-7】查看哪些参数需要延迟生效。

```
SQL> select name
  2  from v$parameter
  3  where ISSYS_MODIFIABLE = 'DEFERRED';

NAME
----------------------------------------------------------------
backup_tape_io_slaves
transaction_auditing
object_cache_optimal_size
object_cache_max_size_percent
sort_area_size
sort_area_retained_size
```

```
已选择 6 行。
```

为了验证 deferred 参数，我们测试对参数 sort_area_size 的修改，先不使用 deferred 参数，看系统如何反应。

【实例 14-8】测试修改参数 sort_area_size 的值。

```
SQL> alter system set sort area size = 65536;
alter system set sort_area_size = 65536
                                *
ERROR 位于第 1 行:
ORA-02096: 此选项的指定初始化参数不可修改
```

显然 Oracle 系统不允许修改该参数，所以必须使用 deferred 参数，如实例 14-9 所示。

【实例 14-9】使用 deferred 参数修改 sort_area_size 的值。

```
SQL> alter system set sort_area_size = 65536 deferred scope = spfile;

系统已更改。
```

参数 scope = spfile 的含义是把修改的参数值保存在 SPFILE 文件中。我们再使用实例 14-10 验证当前会话的 sort_area_size 是否已经修改。

【实例 14-10】查询当前会话的 sort_area_size 是否已经修改。

```
SQL> show parameter sort

NAME                                 TYPE        VALUE
------------------------------------ ----------- --------------------
nls_sort                             string
sort_area_retained_size                integer     0
sort_area_size                       integer     524288

```

从上述结果可以看出，虽然我们在实例 14-9 中已经更改了参数 sort_area_size 的值，但是在当前会话中查询时，却没有更改，这证明了该参数的延迟修改效果。

下面关闭数据库，并重新启动数据库，再次查询该参数，如实例 14-11 所示。

【实例 14-11】重新启动数据库，并再次查询该参数 sort_area_size 的值。

```
SQL> shutdown immediate              //关闭数据库
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL> conn /as sysdba
已连接到空闲例程。
SQL> startup                         //重启数据库
ORACLE 例程已经启动。

Total System Global Area  539856896 bytes
Fixed Size                 1334380 bytes
```

```
Variable Size           247464852 bytes
Database Buffers         281018368 bytes
Redo Buffers             10039296 bytes
数据库装载完毕。
数据库已经打开。
SQL> show parameter sort_area_size     //再次查询参数 sort_area_size

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
sort area size                        integer     65536
```

此时发现参数 sort_area_size 的值已经更改了。对参数的说明如下。

- <scope = memory|spfile|both>：说明把修改的参数保存的位置，其中 memory 说明把参数保存在内存中，重启数据库实例时该参数无效，SPFILE 说明把参数值保存在 SPFILE 中，重启数据库后仍有效，both 表明把参数保存在内存和 SPFILE 中。
- <sid = 'sid|*'>：该参数用于集群系统，默认值为 sid='*'，其作用是为集群中所有的实例指定唯一的参数设置。如果不使用 RAC，则没有必要使用该设置。

14.3.4　取消 SPFILE 中的参数值

既然可以设置 SPFILE 文件中的值，也应该有某种方式取消参数值。因为 SPFILE 是二进制文件，所以无法使用文本编辑器编辑。Oracle 允许使用 ALTER SYSTEM 指令取消 SPFILE 中的参数值。其语法格式如下。

```
ALTER SYSTEM RESET parameter <scope = memory|spfile|both> sid = 'sid|*'
```

采用这种方式将使得该参数保持原来的默认值，而不是用户设置的参数值。给出实例 14-12 说明更改参数值的方法。

【实例 14-12】更改参数值。

```
SQL> alter system reset sort area size scope = spfile sid ='*';

系统已更改。
```

此时在 SPFILE 文件中不会再有 sort_area_size 参数，该参数将采用默认值。

在 Oracle 11g 中允许使用从内存创建 PFILE 和 SPFILE 的方式，这样就可以获得当前运行的数据库的真实参数值。下面通过实例演示如何从内存创建 PFILE，前提是保证数据库至少处于 NOMOUNT 状态。

【实例 14-13】登录数据库。

```
C:\Documents and Settings\Administrator>sqlplus /nolog

SQL*Plus: Release 11.1.0.6.0 - Production on 星期五 10 月 16 16:39:59 2009
```

```
Copyright (c) 1982, 2007, Oracle.  All rights reserved.

SQL> connect /as sysdba
```

【实例 14-14】从内存创建 PFILE。

```
SQL> create pfile = 'd:/pfile/initial.ora' from memory;

文件已创建。
```

现在已成功创建了一个 PFILE 文件，该文件位于目录 d:/pfile 下，文件名为 initial.ora。下面关闭数据库，从 PFILE 启动数据库。

【实例 14-15】通过 PFILE 启动数据库

```
SQL> startup pfile = 'd:/pfile/initial.ora'
ORACLE 例程已经启动。

Total System Global Area  539856896 bytes
Fixed Size                  1334380 bytes
Variable Size             281019284 bytes
Database Buffers          247463936 bytes
Redo Buffers               10039296 bytes
数据库装载完毕。
数据库已经打开。
```

现在数据库是通过 PFILE 打开的，我们可以通过参数 spfile 的值验证是否是使用 PFILE 启动数据库的。

【实例 14-16】验证是否使用 PFILE 启动数据库。

```
SQL> show parameter spfile;

NAME                                 TYPE        VALUE
-------------------------------------------------------- -
spfile                               string
```

上述输出的参数 spfile 的值为空，说明数据库不是通过 SPFILE 启动的。同样可以从内存中创建 SPFILE，如下例所示。

【实例 14-17】内存中创建 SPFILE。

```
SQL> create spfile from memory;

文件已创建。
```

14.4　启动数据库实例

数据库启动时要读取参数文件，不然数据库无法正常启动，但是它如何选择初始化参数文件呢？下面将依次讲解数据库启动时的初始化参数选择规则、简单的故障现象以及如何处理。

数据库实例启动时，会按照以下规则寻找数据库初始化参数文件。

01 使用服务器上的 spfile<ORACLE_SID>.ora 文件启动数据库。
02 选择服务器上默认的 SPFILE 文件启动。
03 如果没找到 SPFILE，就将服务器上的 init<ORACLE_SID>.ora 文件作为启动参数文件。
04 如果没有找到 init<ORACLE_SID>.ora 文件，则使用服务器上默认的 PFILE 来启动数据库。

上述启动规则是在用户输入 STARTUP 指令后，数据库搜索初始化参数文件的过程。也可以使用 PFILE 参数，直接指定启动数据库的初始化参数文件，如实例 14-18 所示，直接使用静态初始化文件作为启动数据库的初始化参数文件。

【实例 14-18】使用 init<ORACLE_SID>.ora 文件启动数据库。

```
SQL> startup pfile = 'd:\init_2009_06_11.ora
ORACLE 例程已经启动。

Total System Global Area  118255568 bytes
Fixed Size                 282576 bytes
Variable Size             83886080 bytes
Database Buffers          33554432 bytes
Redo Buffers               532480 bytes
数据库装载完毕。
数据库已经打开。
```

【实例 14-19】数据库初始化参数文件故障。

```
SQL> startup
ORA-01078: failure in processing system parameters
LRM-00109: could not open parameter file
'C:\ORACLE\ORA90\DATABASE\INITLIN.ORA'
```

上述告警的含义是在处理系统参数时失败，原因是无法打开一个参数文件，显然根据 startup 启动数据库时的默认选择过程，它无法找到一系列的启动参数，所以数据库启动失败。遇到这种问题可以通过前面介绍的方法重建 PFILE 文件，再重新启动数据库。

14.5　使用告警文件监督实例

数据库实例启动过程的信息会存储在告警日志文件中，我们可以使用告警日志文件中的参数恢复 SPFILE 文件。当然也可以通过该文件查看故障信息。

告警日志文件的默认文件名为<ORACLE_SID>ARLT.ora，默认存储目录为$ORACLE_HOME\admin\ORACLE_SID\bdump。在 Oracle 11g 中告警追踪文件的名字为 alert_<ORACLE_SID>，默认存储目录为 ORACLE_BASE\diag\rdbms\orcl\orcl\trace。在笔者的计算机上 ORACLE_SID 为 orcl，所以默认文件名为 alert_orcl.ora，而默认存储目录为 ORACLE_BASE\diag\rdbms\orcl\orcl\trace，如图 14-6 所示。

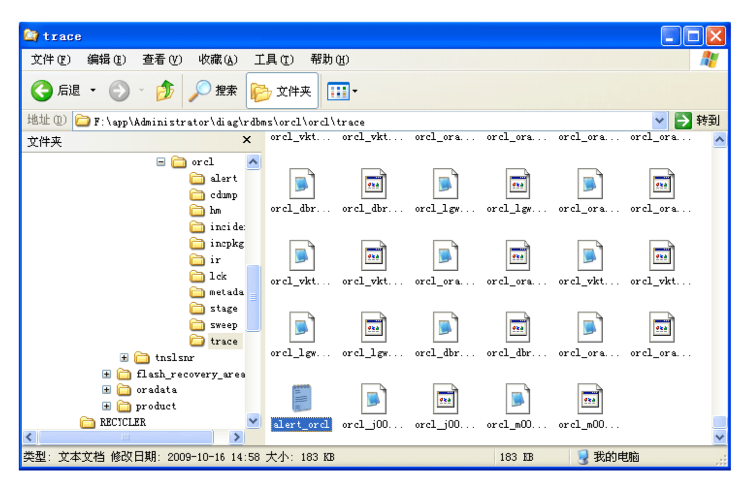
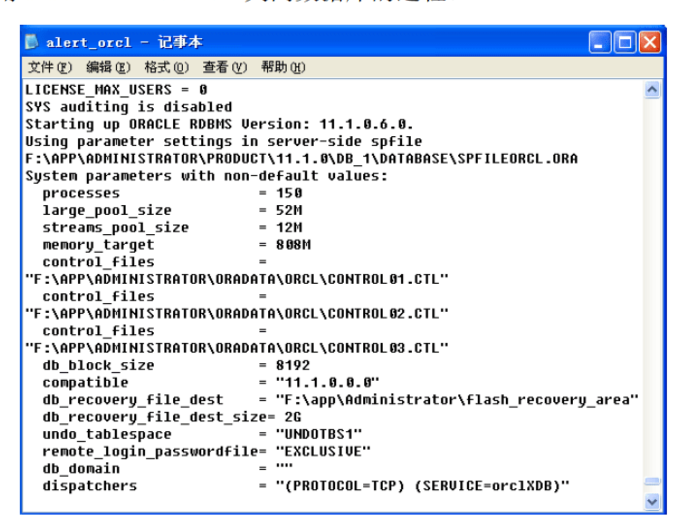


图 14-6　告警日志文件的位置

告警日志文件是一个文本文件，可以利用记事本等工具打开，如图 14-7 所示，这部分的日志文件记录了使用 shutdown immediate 关闭数据库的过程。



```
alert_orcl - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 11.1.0.6.0.
Using parameter settings in server-side spfile
F:\APP\ADMINISTRATOR\PRODUCT\11.1.0\DB_1\DATABASE\SPFILEORCL.ORA
System parameters with non-default values:
  processes                = 150
  large_pool_size          = 52M
  streams_pool_size        = 12M
  memory_target            = 808M
  control_files            =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL01.CTL"
  control_files            =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL02.CTL"
  control_files            =
"F:\APP\ADMINISTRATOR\ORADATA\ORCL\CONTROL03.CTL"
  db_block_size            = 8192
  compatible               = "11.1.0.0.0"
  db_recovery_file_dest    = "F:\app\Administrator\flash_recovery_area"
  db_recovery_file_dest_size= 2G
  undo_tablespace          = "UNDOTBS1"
  remote_login_passwordfile= "EXCLUSIVE"
  db_domain                = ""
  dispatchers              = "(PROTOCOL=TCP) (SERVICE=orclXDB)"
```

图 14-7　告警日志文件内容

在告警日志文件中记录了一些重要事件和指令信息，这些信息如下所示。

- 启动和关闭数据库的具体事件，精确到秒。
- 所有 ALTER 语句。
- 所有非默认的系统初始化参数。
- 创建的表空间和还原段等。

告警日志文件包括一系列的数据库事件，如数据库启动与关闭的详细过程、数据库状态改变等。经常查看告警日志文件是 DBA 的良好习惯。在 Oracle 中告警日志文件的存储目录通过一个参数维护，该参数是 BACKGROUND_DUMP_DEST。可使用数据字典 v$parameter 查询该参数

的值，如实例 14-20 所示。

【实例 14-20】使用数据字典 v$parameter 查询参数 background_dump_dest 的值。

```
SQL> show parameter background dump dest;

NAME                                 TYPE        VALUE
------------------------------------ -----------
background_dump_dest                 string      f:\app\administrator\diag\rdbm
s\orcl\orcl\trace
```

14.6　本章小结

本章介绍了数据库实例和参数文件的关系，强调了参数文件的重要性，介绍了管理 Oracle 初始化参数和参数文件的基础知识，了解了如何设置参数、查看参数值，以及如何让这些设置在数据库重启时仍然有效。分析了两类数据库参数文件：静态参数文件的 PFILE（简单的文本文件）和动态参数文件 SPFILE（服务器参数文件）。但是，推荐使用 SPFILE，而不要使用 PFILE（除非 SPFILE 损坏），因为这更易于管理，而且也更为简洁。使用 ALTER SYSTEM 命令，可以动态地改变参数值，并且可以永久保留在 SPFILE 文件中。

第 15 章

表空间与数据文件管理

为了管理数据文件，Oracle 提出了表空间的概念，本章将讲解 Oracle 引入的逻辑结构和物理结构、二者之间的关系，以及 DBA 需要熟练操纵的表空间管理，读者通过本章的学习，应该掌握如何创建表空间（数据字典管理和本地管理）、如何设置表空间为脱机或只读状态，以及如何修改表空间中数据文件的大小等。

15.1 逻辑结构和物理结构

Oracle 数据库系统具有跨平台特性，在一个数据库平台上开发的数据库可以不加修改地移植到另一个操作系统平台上。这样 Oracle 就不会直接操作底层操作系统的数据文件，而是提供一个中间层，这个中间层就是 Oracle 的逻辑结构，它与操作系统的平台无关，而中间层到数据文件的映射通过 DBMS 来完成，如图 15-1 所示。

如图 15-1 所示，Oracle 数据库应用系统通过操作中间件来实现逻辑操作，而逻辑操作到数据文件操作之间是通过 DBMS 来映射完成的。这样数据文件对于 Oracle 数据库应用系统是透明的。

我们再给出 Oracle 为了管理数据文件而引入的逻辑结构，该逻辑结构也正是图 15-1 中的中间件的内容，如图 15-2 所示。该图展示了数据文件管理的逻辑结构和物理结构。图的左边是逻辑结构。逻辑结构从上到下是包含关系，也是一对多的关系，即一个数据库有一个或多个表空间，一个表空间有一个或多个段，而一个段由一个或多个区段组成，一个区段由多个数据库块组成，一个数据库块由多个操作系统数据库块组成。

右边是物理结构，一个表空间有一个或多个数据文件，一个数据文件物理上由操作系统块组成。

1. 逻辑结构

下面详细解释逻辑结构的各种组成。

- 表空间（Tablespace）：在逻辑上一个数据库由表空间组成，一个表空间只能属于一个数据库，而反之不成立，一个表空间包括一个或多个操作系统文件，把这些操作系统文件称为数据文件。表空间包含一个或多个段。

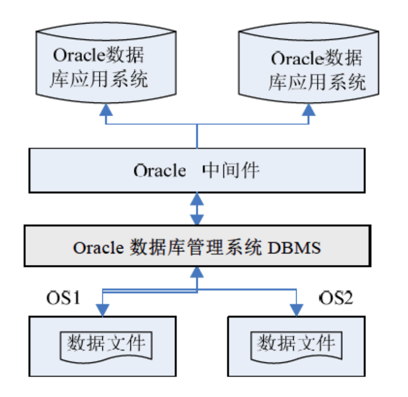


图 15-1　Oracle 的跨平台特性

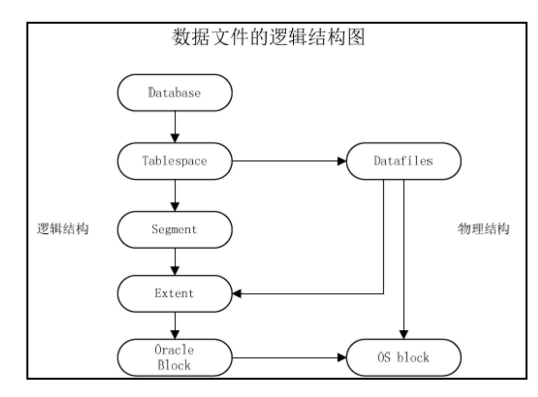


图 15-2　数据文件管理的逻辑结构图

- 段（Segment）：段是表空间内的一个逻辑存储空间，一个表空间包含一个或多个段，一个段不能跨越表空间，即一个段只能在一个表空间中，但是段可以跨越数据文件，即一个段可以分布在同一个表空间的几个数据文件上。一个段由一个或多个区段组成。
- 区段（Extent）：区段是段中分配的逻辑存储空间，一个区段由连续的 Oracle 数据库块组成，一个区段只能存在于一个数据文件中。但创建一个段时，该段至少包含一个区段，但段增长时，将分配更多的区段给该段，同时 DBA 可以手动地向段中添加区段。
- 数据库块（Oracle Block）：Oracle 数据库服务器管理存储空间中的数据文件的最小单位就是数据库块，它是 Oracle 数据库系统输入、输出的最小单位。一个数据库块由一个或多个操作系统块组成。Oracle 提供了标准的数据库块尺寸，该尺寸通过初始化参数 DB_BLOCK_SIZE 设置，在初始创建数据库时指定。一个数据库块应该是操作系统数据库块的整数倍，这样可以避免不必要的 I/O。数据库块在创建数据库时由初始化参数 DB_BLOCK_SIZE 设置，一般大小为 4K 或 8K。可以使用实例 15-1 来查看各个表空间所使用的数据库块的大小。

【实例 15-1】查看表空间的数据库块大小。

```
SQL> select tablespace_name,block_size,contents
  2  from dba_tablespaces;

TABLESPACE_NAME                    BLOCK_SIZE CONTENTS
------------------------------------------------------------------ -----------------------
SYSTEM                                  4096      PERMANENT
UNDOTBS                                 4096     UNDO
CWMLITE                                 4096     PERMANENT
DRSYS                                   4096      PERMANENT
EXAMPLE                                 4096    PERMANENT
INDX                                    4096      PERMANENT
TEMP                                    4096     TEMPORARY
TOOLS                                   4096     PERMANENT
USERS                                   4096     PERMANENT
RBS                                     4096      PERMANENT
```

```
已选择 10 行。
```

实例 15-1 的输出结果说明该数据库中的表空间的数据库块尺寸为 4K（4096 字节）。

2. 物理结构

数据文件物理结构的各组成部分如下。

- 数据文件（datafile）：数据文件是 Oracle 格式的操作系统文件，即.dbf 文件等。数据文件的大小决定了表空间的大小，当表空间不足时，需要增加新的数据文件或者重新设置当前数据文件的大小，以满足表空间增长的需求。
- 操作系统块（OS Block）：操作系统块依赖于不同的操作系统平台，它是操作系统操作数据文件的最小单位，一个或多个操作系统块组成了一个数据库块。

15.2 表空间的分类

在一个数据库中表空间的数量没有严格限制，大小为 2G 的表空间和大小为 20M 的表空间可以并存，只是用户根据业务需求赋予的表空间功能不同（系统表空间除外）。有几个表空间是所有 Oracle 数据库必备的表空间，它们是 System 表空间、临时表空间、还原表空间和默认表空间，在 Oracle 11g 中还有 Sysaux 表空间，该表空间是 System 表空间的扩充，包含各种 Oracle 产品和功能部件使用的数据。虽然还原表空间、默认表空间以及临时表空间可以使用 System 表空间，但是这些表空间是必须的。

Oracle 数据库把表空间分为两类：系统表空间和非系统表空间。

- 系统表空间，顾名思义是数据库系统创建时需要的表空间，这些表空间在数据库创建时自动创建，是每个数据库必须的表空间，满足数据库系统运行的最低要求，如系统表空间（SYSTEM）中存放数据字典或者存放还原段。在用户没有创建非系统表空间时，系统表空间可以存放用户数据或索引，但是这样做会增加系统表空间的 I/O，影响系统效率。
- 非系统表空间是用户根据业务需求而创建的表空间，非系统表空间可以按照数据多少、使用频度、需求数量等方面灵活地设置，这些表空间可以存储还原段或临时段，即创建还原表空间和临时表空间（默认是在系统表空间中），这样一个表空间的功能就相对独立，在特定的数据库应用环境下可以很好地提高系统的效率。通过创建用户自定义的表空间，如还原表空间、临时表空间、数据表空间或者索引表空间，使得数据库的管理更加灵活、方便。

15.3 表空间的区段管理

Oracle 提供了两种管理表空间的区段的方案，一种是数据字典管理，另一种是本地管理，其实管理表空间的实质就是为用户分配可用的区段以及回收空闲区段的过程。两种管理表空间的方

式体现在对表空间区段的管理方式的不同上，造成系统的效率也有所不同。Oracle 推荐使用本地管理表空间的方式，在 Oracle 9i 及以上版本中默认创建的表空间是本地管理的。

15.3.1　数据字典管理的表空间

我们已经讲过表空间的管理就是对该表空间区段的管理，但用户插入数据或表中的其他对象，如索引等增加时，就需要分配更多的区段给新增的数据使用。

数据字典管理的方式是将每个数据字典管理的表空间的使用情况记录在数据字典的表中，当分配或撤销表空间区段的分配时，则隐含使用 SQL 语句对表操作以记录当前表空间区段的使用情况，并且在还原段中记录了变换前的区段使用情况，就像操作普通表时的行为一样，显然这种方式增加了数据字典的频繁操作，对于一个大型的数据库系统，有几百个甚至上千个表空间需要管理，可以想象这样的系统效率将非常低下。

Oracle 已经注意到数据字典管理表空间的问题，所以引入了本地管理的表空间。

15.3.2　本地管理的表空间

本地管理的表空间是为了解决数据字典管理表空间效率不高的问题，Oracle 设计让每一个表空间自己管理表空间区段的分配，记录区段的使用情况。

在数据文件头中有一个区域用于存储本地管理的表空间的数据文件的空间信息，将表空间中数据文件的可用和已用空间信息记录下来。显然这样的管理方式类似于一种分布式管理，减轻了数据字典的工作压力。

本地管理的方式使用位图在数据文件头中记录数据文件的可用和已用信息，位图使用一个数据位表示一个数据库块或者一组数据库块的使用情况，而表空间分配的最小单位为区段 EXTENT，而一个区段由多个数据库块组成，所以当需要在表空间中增加新对象时，就需要查找位图，看是否有一段连续的 Oracle 数据库块空闲。

显然，不使用数据字典管理的表空间提高了系统效率，解决了数据字典的瓶颈问题，但是任何事物都有两面，使用本地管理的表空间不能随意更改默认的存储参数，如初始区段的大小、最大区段数等，但是效率的提高足以弥补本地管理表空间的不足，所以，Oracle 极力推荐使用本地管理的表空间。

15.4　表空间的创建

在一个生产数据库中，往往存在大量的表空间，根据业务需要将用户表或其他对象保存在表空间中，从而根据硬件环境来减少数据库的 I/O，也方便数据空间的维护。本节将通过实例演示如何创建数据字典管理的表空间和本地管理的表空间。

15.4.1 创建表空间概述

创建表空间的语法为：

```
CREATE TABLESPACE tablespac_ename
[DATAFILE clause ]
[MINIMUM EXTENT integer[k|m]]
[BLOCKSIZE integer[k]]
[LOGGING|NOLOGGING]
[DEFAULT storage_clause]
[ONLINE|OFFLINE]
[PERMANENT|TEMPORARY]
```

下面依次解释这些子句的含义。

- DATAFILE 子句：组成该表空间的数据文件的名字，该子句应该给出完整的目录和文件名，目录必须存在，否则无法创建成功。
- MINIMUM EXTENT：定义该表空间中最小的区段大小，这样该表空间中的区段大小为该最小值的整数倍。
- BLOCKSIZE：指出该表空间使用的非标准块尺寸的大小，单位为 K。在设置该参数前必须相应的先修改参数文件中的两个参数，即 DB_CACHE_SIZE 和 DB_nK_CACHE_SIZE，并且这两个参数的值必须与 BLOCKSIZE 的值相同。但使用默认标准块尺寸时，参数 DB_nK_CACHE_SIZE 的值为 0。
- [LOGGING|NOLOGGING]：该参数说明是否把该表空间中数据的变化记录在重做日志文件中。LOGGING 为记录变化，NOLOGGING 为不记录变化。
- DEFAULT 子句：该子句定义该表空间的一些参数，如初始区段的尺寸、最小区段尺寸、最大区段数量等。
- [ONLINE|OFFLINE]：该参数指出该表空间创建后是否联机，ONLINE 为创建后立即联机，OFFLINE 为创建后不联机。默认为联机状态。
- [PERMANENT|TEMPORARY]：参数 PERMANENT 表示该表空间只能存储永久对象，TEMPORARY 说明该表为临时表空间，该表空间只能用于临时存储非永久对象，如用户使用 ORDER BY 查询数据时的排序结果，临时表空间不需要将变化记录到重做日志文件，它只包含用户会话期间的数据，如排序中间结果等。不使用该参数则默认为永久表空间。

下面通过实例 15-2 说明如何创建表空间：创建一个表空间，表空间名为 user_data，该表空间用来存储用户表，表空间就包含一个数据文件，大小为 100M，文件名为 d:\userdata\userdata1.dbf。

【实例 15-2】创建表空间 user_data。

```
SQL> create tablespace user_data
  2  datafile 'd:\userdata\userdata1.dbf' size 100 M
表空间已创建。
```

此时查看该表是否创建，如实例 15-3 所示。

【实例 15-3】查看该表空间 user_data 是否创建。

```
SQL> select tablespace name,logging,stat
  2  from dba_tablespaces;

TABLESPACE_NAME      LOGGING   STATUS
-------------------------------------- ----------------- -----------
SYSTEM                     LOGGING    ONLINE
UNDOTBS                   LOGGING    ONLINE
CWMLITE                   LOGGING    ONLINE
DRSYS                      LOGGING    ONLINE
EXAMPLE                   LOGGING    ONLINE
INDX                       LOGGING     ONLINE
TEMP                       NOLOGGING ONLINE
TOOLS                      LOGGING    ONLINE
USERS                      LOGGING    ONLINE
RBS                         LOGGING    ONLINE
USER_DATA                 LOGGING    ONLINE

已选择 11 行。
```

表空间 USER_DATA 已创建成功，且处于记录日志模式，该表空间目前处于在线状态。接着，我们查看该表空间对应的数据文件是否存在，如实例 15-4 所示。

【实例 15-4】查看表空间 user_data 对应的数据文件。

```
SQL> select file_name,tablespace_name,status
  2  from dba_data_files
  3  where tablespace name = 'USER DATA';

FILE_NAME                    TABLESPACE_NAME      STATUS
------------------------------------------------ --------
D:\USERDATA\USERDATA1.DBF   USER_DATA            AVAILABLE
```

显然，表空间 USER_DATA 中包含一个数据文件，该数据文件位于目录 D:\USERDATA 下，文件名为 USERDATA1.DBF。

这里没有演示创建表空间的全部参数，因为在有些条件下某些参数是不能创建的。

15.4.2　创建数据字典管理的表空间

创建一个数据字典管理的表空间，该表空间有三个数据文件，分别存放在不同的磁盘以及目录下，这样做的目的是有利于平衡 I/O，每个数据文件的大小为 100M，最小区段为 20K，默认存储参数为：初始区段大小为 20K，当再次分配区段（EXTENT）时，分配的区段（EXTENT）大小也为 20K，所分配的最大磁盘空间为 500 个区段（EXTENT），如实例 15-5 所示。

【实例 15-5】创建数据字典管理的表空间。

```
SQL> create tablespace tianjin_data
```

```
  2  datafile 'd:\userdata\tianjin01.dbf' size 100M,
  3        'e:\userdata\tianjin02.dbf' size 100M,
  4        'f:\userdata\tianjin03.dbf' size 100M
  5  minimum extent 20k
  6  extent management dictionary
  7  default storage(initial 20k next 20k maxextents 500 pctincrease 0);

表空间已创建。
```

此时，成功创建了一个表空间，该表空间有三个数据文件，分别存放在不同的磁盘上，下面通过实例 15-6 来验证创建结果。

【实例 15-6】验证实例 15-5 中创建的表空间。

```
SQL> select tablespace name,extent management
  2  from dba_tablespaces
  3  where tablespace_name ='TIANJIN_DATA ';

TABLESPACE_NAME                 EXTENT_MAN
-------------------------------------------------------- ---------------------
TIANJIN_DATA                    DICTIONARY
```

此时在数据字典 DBA_TABLESPACES 中可以查询到实例 15-5 创建的数据字典，并且其区段管理方式为 DICTIONARY。

在表空间 TIANJIN_DATA 中，我们创建了三个数据文件，分别存放在不同的磁盘上，下面通过实例 15-7 来验证这些数据文件的信息。

【实例 15-7】验证表空间 TIANJIN_DATA 中的数据文件。

```
SQL> col file_name for a30
SQL> col tablespace_name for a20
SQL> run
  1  select tablespace_name,file_name,blocks,status
  2  from dba_data_files
  3* where tablespace name = 'TIANJIN DATA'

TABLESPACE_NAME     FILE_NAME              BLOCKS STATUS
------------------------------------ ----------------------
TIANJIN_DATA     D:\USERDATA\TIANJIN01.DBF     25600 AVAILABLE
TIANJIN_DATA     E:\USERDATA\TIANJIN02.DBF     25600 AVAILABLE
TIANJIN_DATA     F:\USERDATA\TIANJIN03.DBF     25600 AVAILABLE
```

实例 15-7 的输出说明，数据字典 TIANJIN_DATA 有三个数据文件，且数据文件的状态都为 AVAILABLE 的。

在实例 15-5 中使用了一些默认存储参数，下面再用实例 15-8 来验证这些参数。

【实例 15-8】查询表空间 TIANJIN_DATA 的默认存储参数。

```
SQL> set line 100
SQL>select tablespace_name,block_size,initial_extent,next_extent,max_extents,
pct_increase
```

```
  2  from dba_tablespaces
  3* where tablespace_name = 'TIANJIN_DATA'
TABLESPACE NAME   BLOCK SIZE INITIAL EXTENT NEXT EXTEN MAX EXTENTS PCT INCREASE
---------------------------------- ------------------ --------------------- -------
TIANJIN_DATA              4096         20480      20480        500            0
```

实例 15-8 的输出说明，表空间 TIANJIN_DATA 第一次分配区段时，区段的大小为 20480 字节，即 20K，再次分配区段时，区段大小也为 20480 字节，即 20K，最大区段数量为 500 个区段（EXTENT）。

15.4.3　创建本地管理的表空间

创建一个本地管理的表空间，该表空间名为 BEIJING_DATA，由于本地管理的表空间不能随意更改存储参数，所以创建起来较之创建数据字典管理的表空间要简洁一些，如实例 15-9 所示。

【实例 15-9】创建本地管理的表空间。

```
SQL> create tablespace beijing_data
  2  datafile 'd:\userdata\beijingdata01.dbf' size 100M
  3  extent management local
  4  uniform size 1M;

表空间已创建。
```

在该实例中，创建了名为 BEIJING_DATA 的表空间，该表空间只有一个数据文件，大小为 100M，区段（EXTENT）管理方式为本地管理（LOCAL），区段尺寸统一为 1M。下面通过实例 15-10 来验证表空间 BEIJING_DATA 的区段管理方式。

【实例 15-10】验证表空间 BEIJING_DATA 的区段管理方式。

```
SQL> select tablespace_name,block_size,extent_management,status
  2  from dba_tablespaces
  3* where tablespace_name like 'BEIJING%'

TABLESPACE_NAME      BLOCK_SIZE EXTENT_MAN STATUS
-------------------- ---------- ---------- ---------
BEIJING_DATA               4096 LOCAL      ONLINE
```

实例 15-10 的输出结果表明，表空间 BEIJING_DATA 为本地管理，因为其 EXTENT_MAN 为 LOCAL，且默认该表空间一旦创建就是联机状态，所以 STATUS 为 ONLINE。

【实例 15-11】验证表空间 BEIJING_DATA 的数据文件信息。

```
SQL> select tablespace_name,file_name, status
  2  from dba_data_files
  3  where tablespace_name ='BEIJING_DATA';

TABLESPACE_NAME FILE_NAME                          STATUS
-------------------------------------- ----------------
BEIJING_DATA     D:\USERDATA\BEIJINGDATA01.DBF   AVAILABLE
```

输出说明新建的表空间 BEIJING_DATA 中只有一个数据文件，该文件存储在目录 D:\USERDATA 下，文件名为 BEIJING_DATA01.DBF。在建立本地管理的表空间时，我们没有使用默认存储参数，只是使用了一个 UNIFORM SIZE 参数，设置统一的区段尺寸，下面通过实例 15-12 来验证该本地管理的表空间的存储参数信息。

【实例 15-12】查看本地管理的表空间的存储参数信息。

```
SQL> select tablespace_name,block_size,initial_extent,next_extent,max_extents,
pct_increase
  2  from dba tablespaces
  3  where tablespace_name = 'BEIJING_DATA';

TABLESPACE_NAME BLOCK_SIZE INITIAL_EXTENT NEXT_EXTENT MAX_EXTENTS PCT_INCREASE
------------------------------ ---------- -------------- ----------- -----------
BEIJING_DATA   4096       1048576        1048576       2147483645          0
```

表空间 BEIJING_DATA 的初始区段大小为 1M，再次分配区段时区段大小也为 1M，数据库块尺寸为默认标准块尺寸 4K 字节。

15.4.4 创建还原表空间

还原表空间用于存放还原段。这里通过实例说明还原段的作用，如果一个用户要修改某个属性值，把月薪金额从 2000 更改到 2500，在更改的过程中其他用户要查看该数据时，看到的应该是 2000，因为当前用户正在更改数据，还没有提交，所以为了保证这种读的一致性，Oracle 设计了还原段，在还原段中存放更改前的数据。

还原表空间只能存放还原段，不能存放其他任何对象。在创建还原表空间时，只能使用 DATAFILE 子句和 EXTENT MANAGEMENT 子句。通过实例 15-13 演示如何创建还原表空间。

【实例 15-13】创建还原表空间 USER_UNDO。

```
SQL> create undo tablespace user_undo
  2  datafile 'd:\userundo\user_undo.dbf'
  3* size 30M

表空间已创建。
```

同样，通过实例 15-14 验证是否成功创建还原表空间 USER_UNDO。

【实例 15-14】查看是否成功创建还原表空间 USER_UNDO。

```
SQL> select tablespace name,status,contents,logging,extent manageme
  2  from dba_tablespaces;

TABLESPACE_NAME  STATUS    CONTENTS  LOGGING   EXTENT_MAN
------------------------------ ---------------- ------------------ -----
SYSTEM                 ONLINE    PERMANENT LOGGING   DICTIONARY
UNDOTBS               ONLINE    UNDO       LOGGING   LOCAL
CWMLITE               ONLINE    PERMANENT LOGGING   LOCAL
```

```
DRSYS                ONLINE    PERMANENT LOGGING   LOCAL
EXAMPLE              ONLINE    PERMANENT LOGGING   LOCAL
INDX                 ONLINE    PERMANENT LOGGING   LOCAL
TEMP                 ONLINE    TEMPORARY NOLOGGING LOCAL
TOOLS                ONLINE    PERMANENT LOGGING   LOCAL
USERS                ONLINE    PERMANENT LOGGING   LOCAL
RBS                  ONLINE    PERMANENT LOGGING   LOCAL
USER_UNDO            ONLINE    UNDO      LOGGING   LOCAL

已选择 11 行。
```

在上述输出中，从最后一行可以看出表空间 USER_UNDO 已经创建，该表空间的状态为联机状态，CONTENTS 为 UNDO，说明它是还原表空间，LOGGING 说明该表空间的变化受重做日志的保护，区段的管理方式为本地管理。

下面通过实例 15-15 来验证新创建的还原表空间的存储参数设置信息。

【实例 15-15】查看还原表空间 USER_UNDO 的存储参数。

```
SQL> select tablespace_name,block_size,initial_extent,next_extent,max_extents
  2  from dba_tablespaces
  3  where contents = 'UNDO';

TABLESPACE_NAME  BLOCK_SIZE INITIAL_EXTENT NEXT_EXTENT MAX_EXTENTS
-------------------------------- ---------------- ---------
UNDOTBS                     4096         65536            2147483645
USER_UNDO                   4096         65536            2147483645
```

我们查询了内容为 UNDO 的表空间信息，查询结果说明当前的数据库有两个还原表空间，其中 UNDOTBS 是系统创建的，其实在 Oracle 10g 中虽然只有 SYSTEM 和 SYSAUX 表空间是强制创建的，但是在安装时会默认创建一个 UNDOTBS1 的还原表空间、一个还原数据文件，默认文件名为 UNDOTBS01.DBF。在实例 15-15 中表空间 USER_UNDO 是刚刚创建的，它的默认数据库块尺寸为 4096 字节，初始区段大小为 65536 字节，而可分配的最大区段不是一个数量值，和数据字典管理的表空间不同，而是一个数据位数，因为本地管理的表空间是通过数据位记录空间的数据库块的使用情况。

在创建还原表空间时，创建了一个数据文件，下面通过实例 15-16 查看还原表空间中数据文件的信息。

【实例 15-16】查看还原表空间 USER_UNDO 的数据文件。

```
SQL> select file_name,file_id,tablespace_name,status
  2  from dba_data_files
  3* where tablespace_name = 'USER_UNDO'

FILE_NAME                FILE_ID TABLESPACE_NAME  STATUS
------------------------------------------ ----------- ----------
D:\USERUNDO\USER_UNDO.DBF   9     USER_UNDO        AVAILABLE
```

还原表空间 USER_UNDO 中的数据文件为 D:\USERUNDO\USER_UNDO.DBF，该文件当前

可以使用，因为 STATUS 为 AVAILABLE。

15.4.5 创建临时表空间

在 Oracle 数据库中临时表空间用于用户的特定会话活动，如用户会话中的排序操作，排序的中间结果需要存储在某个区域，这个区域就是临时表空间，临时表空间的排序段是在实例启动后有第一个排序操作时创建的。如果在创建数据库时没有创建临时表空间，则数据库服务器默认使用 SYSTEM 表空间，显然这样会影响数据库系统的效率，因为 SYSTEM 表空间中存储了数据字典等数据库系统的重要信息。在 Oracle 9i 中会自动创建一个 TEMP 的临时表空间，在 Oracle 10g 中，只有 SYSTEM 和 SYSAUS 表空间是强制建立的，临时表空间会默认创建，名字为 TEMP，在该表空间中会创建一个临时数据文件，文件名为 TEMP01.DBF，它和其他数据文件存放在同一个目录下，该目录为$ORACLE_HOME/ORADATA/ORACLE_ID。如笔者自己的 ORACLE_ID 为 ORCL，所以数据库文件的默认安装目录为 D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL。

临时表空间是使用当前数据库的多个用户共享使用的，临时表空间中的区段在需要时按照创建临时表空间时的参数或管理方式进行扩展。

下面演示在数据库创建完成后，如何创建一个临时表空间，如实例 15-17 所示。

【实例 15-17】创建一个临时表空间 user_temp。

```
SQL> create temporary tablespace user_temp
  2  tempfile 'd:\usertemp\user_temp.dbf' size 20M
  3  extent management local
  4  uniform size 1M;

表空间已创建。
```

在创建临时表空间时，需要使用 CREATE TEMPORARY 告诉数据库服务器该表空间是临时表空间，并且表空间中的数据文件必须使用 TEMPFILE 标识它是临时表空间的数据文件。

在实例 15-17 中，创建的临时表空间名为 USER_TEMP，区段管理方式为本地管理，区段的统一扩展尺寸是 1M。

下面通过实例 15-18 查询是否成功创建临时表空间 USER_TEMP。

【实例 15-18】查询是否成功创建临时表空间 USER_TEMP。

```
SQL> select tablespace_name,status,contents,logging
  2  from dba tablespaces
  3  where tablespace_name like 'USER%';

TABLESPACE_NAME      STATUS    CONTENTS  LOGGING
------------------------------------ ----------------- -----------------
USERS                  ONLINE    PERMANENT LOGGING
USER_TEMP             ONLINE    TEMPORARY NOLOGGING
USER_UNDO             ONLINE    UNDO      LOGGING
```

表空间 USER_TEMP 为临时表空间，因为 CONTENTS 为 TEMPORARY，该表空间处于联

机状态，尤其需要注意，该表空间为 NOLOGGING，即不需要将临时表空间的变化记录到重做日志文件中。

在实例 15-17 中，临时表空间中创建了一个数据文件，该文件 D:\USER_TEMP\USER_TEMP.DBF 的大小为 20M，通过数据字典视图 v$tempfile 来查看该数据文件信息，如实例 15-19 所示。

【实例 15-19】通过数据字典视图 v$tempfile 来查看数据文件信息。

```
SQL> col name for a30
SQL> select file#,status,enabled,bytes,block size,name
  2  from v$tempfile;

FILE# STATUS  ENABLED  BYTES   BLOCK_SIZE NAME
----------------------- ---------------- ------------ ------------------ -
1    ONLINE  READ WRITE 20971520  4096      D:\USERTEMP\USER_TEMP.DBF
```

通过输出结果说明，该临时数据文件为可读、可写的，当前处于联机状态，文件大小为 20M，数据块尺寸为标准尺寸 4K。

临时表空间中的临时数据文件也是.DBF 格式的数据库格式文件，但是这个数据文件和普通的存储表或索引的数据文件有如下不同。

- 临时文件总是处于 NOLOGGING 模式，因为临时表空间中的数据都是中间数据，只是临时存放的，它们的变化不需要记录在重做日志文件中，因为这些变化本身也不需要恢复。
- 临时文件不能设置为只读（read_only）模式。
- 临时文件不能重命名。
- 临时文件不能通过 ALTER DATABASE 创建。
- 临时文件用于只读数据库。
- 介质恢复时不需要临时文件。
- 使用 BACKUP CONTROLFILE 并不产生任何关于临时文件的信息。
- 使用 CREATE CONTROLFILE 不能设置任何与临时文件有关的信息。
- 在初始化参数文件中，有一个参数为 SORT_AREA_SIZE，这是排序区的尺寸大小，为了优化临时表空间中排序操作的性能，最好设置 UNIFORM SIZE 为 SORT_AREA_SIZE 的整数倍。

默认临时表空间是指一旦该数据库启动，则默认使用该表空间作为默认的临时表空间，用于存放用户会话数据与排序操作。默认临时表空间可以在创建数据库时创建，此时使用指令 DEFAULT TEMPORARY TABLESPACE，也可以在数据库创建成功后创建，此时需要事先建立一个临时表空间，再使用 ALTER DATABASE DEFAULT TEMPORARY TABLESPACE 指令更改临时表空间。

如果在数据库创建时，没有建立临时表空间，数据库创建成功后也没有创建临时表空间且更改默认设置，则 SYSTEM 表空间为默认临时表空间，并且 Oracle 认为这是不合理的行为，把这个信息记录在告警文件 ALERT.LOG 中，告警文件信息如图 15-3 所示。由于临时表空间的区段

使用很频繁，会不断地出现磁盘碎片，这样对 SYSTEM 表空间的使用效率有很大影响，所以必须创建临时表空间。

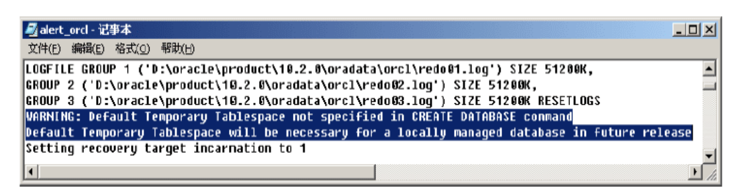


图 15-3　告警文件记录的未设置临时表空间信息

下面，先查看当前数据库的默认临时表空间是哪个表空间，使用静态数据字典 DATABASE_PROPERTIES，它有三个列属性，分别是属性名（PROPERTY_NAME）、属性值（PROPERTY_VALUE）和描述信息（DESCRIPTION），如实例 15-20 所示。

【实例 15-20】查看当前数据库的默认临时表空间。

```
SQL> col property name for a30
SQL> col property_value for a20
SQL> col description for a40
SQL> select *
  2  from database_properties
  3* where property_name like 'DEFAULT%'

PROPERTY NAME            PROPERTY V DESCRIPTION
------------------------------ ----------- ------------------
DEFAULT_TEMP_TABLESPACE TEMP      Name of default temporary tablespace
```

输出显示当前的默认临时表空间为 TEMP，而在 Oracle 10g 中输出略有不同，如实例 15-21 所示。

【实例 15-21】在 Oracle 10g 中查看当前数据库的默认临时表空间。

```
SQL> select *
  2  from database_properties
  3* where property_name like 'DEFAULT%'

PROPERTY_NAME             PROPERTY_VALUE   DESCRIPTION
------------------------------ -------------------- ----------
DEFAULT TEMP TABLESPACE    TEMP          Name of default temporary tablespace
DEFAULT_PERMANENT_TABLESPACE USERS      Name of default permanent tablespace
DEFAULT_TBS_TYPE              SMALLFILE    Default tablespace type
```

在实例 15-21 中，输出多个两行，增加 UERS 为默认永久表空间，用户创建的表或索引如果没有指定表空间，则默认存储在 USERS 中，而且默认的表空间类型为 SMALLFILE（和 Oracle 10g 中的大对象文件类型相对应）。

下面将演示如何将临时表空间切换到 USER_TEMP，在生产数据库中，可能会出现当前的临时表空间不能满足应用需求的情况，DBA 可以创建相应的临时表空间，而后切换为当前使用的临时表空间，切换方法如实例 15-22 所示。

【实例 15-22】切换临时表空间。

```
SQL> alter database default temporary tablespace user_temp;

数据库已更改。
```

输出显示已成功更改默认临时表空间，并通过以下代码来验证更改结果：

```
SQL> select *
  2  from database_properties
  3  where property_name like 'DEFAULT%';

PROPERTY_NAME              PROPERTY_V  DESCRIPTION
-----------------------------------------------  --------------------  ---
DEFAULT_TEMP_TABLESPACE  USER_TEMP   Name of default temporary tablespace
```

此时，当前数据库的默认临时表空间为 USER_TEMP。在用户需要时，默认临时表空间可以随时使用实例 15-22 的指令 ALTER DATABASE DEFAULT TEMPORARY TABLESPACE 进行更改，一旦更改，则所有的用户将自动使用更改后的临时表空间为默认临时表空间。

下面是管理默认临时表空间的一些约束。

- 不能删除一个当前使用的默认临时表空间。在切换到一个新的临时表空间前，当前的默认临时表空间无法删除，如当前数据库的默认临时表空间为 USER_TEMP，如果想删除该表空间，则提示错误，如实例 15-23 所示。
- 不能把默认临时表空间的空间类型改为 PERMANENT，即不能把默认临时表空间改为一个永久 PERMANENT 表空间。
- 不能把默认临时表空间置为脱机状态。表空间处于脱机状态的目的是使得使用这些表空间的应用无法再使用它们，从而完成一些诸如脱机备份、维护等任务。但是类似的操作不会涉及临时表空间，所以 Oracle 设计不能把默认临时表空间置为脱机状态，否则会提示如下错误，如实例 15-24 所示。

【实例 15-23】删除当前使用的临时表空间。

```
SQL> drop tablespace user_temp;
drop tablespace user_temp
*
ERROR 位于第 1 行：
ORA-12906: 不能删除默认的临时表空间
```

若想删除当前使用的默认临时表空间，必须建立一个新的临时表空间，并且切换到该新建立的临时表空间。

【实例 15-24】将默认临时表空间脱机。

```
SQL> alter tablespace user_temp offline;
alter tablespace user_temp offline
*
ERROR 位于第 1 行：
```

```
ORA-03217: 变更 TEMPORARY TABLESPACE 无效的选项
```

15.4.6 创建大文件表空间

大文件表空间是在 Oracle 10g 中提出来的，大文件表空间由一个大文件组成，而不是由多个传统的小文件组成，这使得 Oracle 有能力创建和管理大文件。正是大文件表空间和大文件的一一对应特性，使得表空间成为磁盘空间管理、备份和恢复的操作对象。

1. 使用大文件表空间的一些限制

在 Oracle 10g 中只有本地管理的且段空间自动管理的表空间才能使用大文件表空间（Big File Tablespace），简称大文件表空间为 BTF。但是对于本地管理的回滚表空间和临时表空间，不要求段空间管理的类型，可以使用 BFT。同时在 Oracle 10g CONCEPT 文档中建议，大文件表空间应该和自动存储管理和逻辑卷管理工具结合使用，这些工具能够支持动态扩展逻辑卷，也支持条带化或支持 RIAD。

使用大文件表空间在数据库开启时和与 DBWR 进程的性能相比会有显著提高，但是该表空间的大文件会增加该表空间或整个数据库的备份和恢复时间。

2. 使用大文件表空间的优势

使用大文件表空间的优势如下：

- 只需要创建一个数据文件，大大减少了数据文件的数量，简化了数据文件的管理，显然数据文件的减少使得控制文件的容量也减少，因为控制文件不需要记录大量的数据文件的存储位置信息。
- 大文件表空间的容量比普通表空间要大的多，所以其存储能力有显著提高。一个普通的表空间最多可以用 1024 个数据文件，而一个大文件表空间只包含一个文件，但是此文件的容量上限是普通数据文件的 1024 倍，所以大文件表空间和普通表空间的容量是一样的。但是由于每个数据库最多使用 64k 个表空间，所以使用大文件表空间使得数据库总容量比使用普通表空间时要大得多。根据块的大小，大文件表空间的容量可以为 128，如果最大的数据块为 32K，则数据库最大容量可以达到 8EB。

3. 创建大文件表空间

创建大文件表空间有三种方法，下面依次介绍这三种方法。

（1）方法 1

创建数据库时，定义大文件表空间并把它作为默认表空间。下面给出创建数据库时设置大文件表空间的语句，如下所示：

```
SQL> create database
  1  set default bigfile tablespace tbs_name
  2  datafile 'd:\bigfile tbs\bfile tbs01.dbf' size 2G
…………
```

一旦创建了默认表空间为大文件表空间类型，则以后创建的表空间都为大文件表空间，否则需要手工修改这个默认设置。

（2）方法 2

在数据库成功创建后，使用 CREATE TABLESPACE BIGFILE 子句创建大文件表空间，如实例 15-25 所示。

【实例 15-25】创建大文件表空间 bigfiletbs。

```
SQL> create bigfile tablespace bigfiletbs
  2  datafile 'd:\bigfile_tbs\bfile_tbs01.dbf' size 2G;

表空间已创建。
```

在成功创建了大文件表空间后，再通过实例 15-26 来验证该表空间的一些属性信息。

【实例 15-26】查询表空间的数据文件属性信息。

```
SQL> select tablespace_name,file_name,bytes/(1024*1024*1024) G
  2* from dba_data_files

TABLESPACE_NAME   FILE_NAME                                             G
-------------------- ------------------------------------------------
USERS    F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF       .004882813
SYSAUX    F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF      .244140625
UNDOTBS1 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF     .034179688
SYSTEM   F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF      .46875
EXAMPLE  F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF      .09765625
BIGFILETBS      D:\BIGFILE_TBS\BFILE_TBS01.DBF                          2

已选择 6 行。
```

我们看到表空间 BIGFILETBS 的大小为 2G，唯一的数据文件位于目录 D:\BIGFILE_TBS 下，文件名为 BFILE_TBS01.DBF。

再看一下表空间 BIGFILETBS 的区段管理方式和段空间管理方式，如实例 15-27 所示。

【实例 15-27】查询表空间 BIGFILETBS 的区段管理方式和段空间管理方式。

```
SQL> select tablespace_name,initial_extent,contents,extent_management,
segment_space_management
  2  from dba tablespaces
  3  where tablespace_name like 'BIG%';

TABLESPACE_NAME      INITIAL_EXTENT CONTENTS  EXTENT_MAN SEGMEN
-------------------- -------------- --------- ---------- ------
BIGFILETBS                    65536     PERMANENT  LOCAL      AUTO
```

大文件表空间 BIGFILETBS 的初始区段（INITIAL_EXTENT）大小为 64K，而区段管理方式（EXTENT_MAN）为本地管理方式（LOCAL），段空间管理方式为自动管理方式（AUTO）。

（3）方法 3

通过改变默认表空间为大文件表空间，使得后来创建的表空间都为大文件表空间。Oracle 允许动态地改变当前数据库的默认表空间的类型为大文件表空间或普通表空间。我们用实例说明如何把数据库的默认表空间类型改为大文件表空间类型，如实例 15-28 所示。

【实例 15-28】把数据库的默认表空间类型改为大文件表空间类型。

```
SQL> alter tablespace set default bigfile tablespace
```

4. 更改大文件表空间的大小

在大文件表空间创建后，可以根据需要修改表空间的大小，有两种方式用于实现大文件表空间的容量修改。

（1）方法 1

在 ALTER TABLESPACE 指令中使用 RESIZE 子句，如实例 15-29 所示。

【实例 15-29】更改大文件表空间的尺寸。

```
SQL> alter tablespace bigfiletbs resize 4G;

表空间已更改。
```

我们通过实例 15-30 来验证修改结果。

【实例 15-30】查询更改后的大文件表空间 bigfiletbs 的大小。

```
SQL> col file_name for a30
SQL> select tablespace_name,file_name,bytes/(1024*1024*1024) G,
autoextensible
  2  from dba_data_files
  3* where tablespace_name like 'BIG%'

TABLESPACE_NAME        FILE_NAME                               G AUT
------------------------------ ------------------------------ -------
BIGFILETBS                D:\BIGFILE_TBS\BFILE_TBS01.DBF         4  NO
```

从实例 15-30 的输出可以看出，大文件表空间 BIGFILETBS 的大小已经修改为 4G，注意该表空间的自动扩展方式为 NO，这意味着该表空间不能自动扩展。下面介绍第二种修改大文件表空间的方法。

（2）方法 2

在 ALTER TABLESPACE 指令中使用 AUTOEXTEND ON 子句，如实例 15-31 所示。

【实例 15-31】修改大文件表空间的大小为自动扩展。

```
SQL> alter tablespace bigfiletbs autoextend on next 1G;

表空间已更改。
```

再通过实例 15-32 验证修改结果。

【实例 15-32】验证修改结果。

```
SQL> select tablespace_name,file_name,bytes/(1024*1024*1024) G,autoextensible
  2  from dba_data_files
  3  where tablespace_name like 'BIG%';

TABLESPACE_NAME              FILE_NAME                              G  AUT
------------------------------ ---------------------------------
BIGFILETBS                   D:\BIGFILE_TBS\BFILE_TBS01.DBF     4  YES
```

我们看到该表空间的大小为 4G，当空间不足时扩展方式为自动扩展，因为 AUTOEXTENSIBLE 的值为 YES。

如果不知道当前数据库的默认表空间的类型，可以使用如实例 15-33 所示的方法查看。

【实例 15-33】查询当前数据库的默认表空间的类型。

```
SQL> col property_name for a20
SQL> col property value for a20
SQL> col description for a30
SQL> select *
  2  from database_properties
  3* where property_name ='DEFAULT_TBS_TYPE'

PROPERTY_NAME        PROPERTY_VALUE       DESCRIPTION
-------------------- -------------------- ------------------------------
DEFAULT_TBS_TYPE     SMALLFILE            Default tablespace type
```

从输出可以判断，当前数据库的默认表空间类型为 SMALLFILE，即小文件表空间，也就是普通表空间。

15.5　表空间的管理

本节将从表空间的两种状态、表空间的内容两方面讲解表空间管理的常用知识。

15.5.1　表空间的状态管理

本小节将讲解表空间的脱机管理和只读管理，脱机与只读是表空间的两种状态，在脱机状态下，用户或应用程序无法访问这些表空间，此时可以完成一些如脱机备份等操作，处于只读状态的表空间，用户或应用程序可以访问这些表空间，但是无法更改表空间中的数据，如果一个表空间中的数据不会变化，属于静态数据，这样就可以把相应表空间改为只读，只读表空间不产生变化的数据。

下面依次讲解脱机管理和只读管理。

1. 脱机管理

脱机管理的表空间无法实现数据访问，在 Oracle 数据库中不是所有的表空间都可以设为脱

机管理，其中包括 SYSTEM 表空间、有活跃还原段的表空间和默认临时表空间。

表空间一般是处于联机状态的，此时用户可以访问表空间中的数据，但是有些情况下需要将表空间置于脱机状态，这些情况包括：

- 允许用户访问数据库的一部分，而某些空间不允许用户访问。
- 执行脱机的表空间备份。
- 在数据库打开时，恢复表空间或表空间中的数据文件。
- 在数据库打开时，移动表空间中的数据文件。

当一个表空间处于脱机状态时，Oracle 不允许执行任何的 SQL 语句，用户试图访问存储在该表空间中的对象会报错。当表空间脱机或联机时，这个事件会记录在数据字典和控制文件中。如果关闭数据库时，数据库处于脱机状态，则当数据打开时依然保持脱机状态。



Oracle 实例有时会自动切换表空间到脱机状态，如当数据库写进程尝试向一个表空间中的数据文件写数据，而尝试失败时，则自动将该表空间切换到脱机状态。

下面通过实例说明如何将一个表空间置为脱机状态，在笔者的数据库上曾经创建了 LIN 表空间，我们先查看该表空间的信息，如实例 15-34 所示。

【实例 15-34】查看表空间 lin 的状态。

```
SQL> select status,contents,logging
  2  from dba_tablespaces
  3  where tablespace name = 'LIN';

STATUS    CONTENTS  LOGGING
--------- --------- ---------
ONLINE    PERMANENT LOGGING
```

从输出可以看出该表空间处于联机状态，是永久表空间，用于存储用户表或索引等数据库对象。为了后面演示的需要，我们再看看该表空间中是否存在表，如实例 15-35 所示。

【实例 15-35】查看表空间 LIN 是否存在。

```
SQL> select table_name,owner
  2  from dba_tables
  3  where tablespace name = 'LIN';

TABLE_NAME      OWNER
------------------------------------------
EMPLOYEES        SCOTT
```

可见，表空间 LIN 存储了一个表，表名为 EMPLOYEES，所属用户为 SCOTT。下面将演示如何把表空间 LIN 设置为脱机状态，如实例 15-36 所示。

【实例 15-36】将表空间 LIN 设置为脱机状态。

```
SQL> alter tablespace lin offline;

表空间已更改。
```

下面通过实例 15-37 查看修改结果。

【实例 15-37】查询表空间 LIN 是否脱机。

```
SQL> select status,contents,logging
  2  from dba_tablespaces
  3  where tablespace_name = 'LIN';

STATUS    CONTENTS  LOGGING
--------- --------- ---------
OFFLINE   PERMANENT LOGGING
```

此时，看到该表空间处于离线状态，我们需要确定该表空间中的数据文件的状态，用实例 15-38 进行演示。

【实例 15-38】查询表空间 LIN 中的数据文件的状态。

```
SQL> select file#,name,status
  2  from v$datafile
  3  where file#>10
  4  ;

FILE# NAME                              STATUS
---------- ------------------------------ ----------------------------
11 D:\TEMP\LIN.DBF                       OFFLINE
```

实例 15-37 和实例 15-38 说明，表空间 LIN 和其中的数据文件都处于脱机状态，这样用户就无法查询该表空间中的数据库对象，否则会提示错误，我们查询表空间 LIN 中的表 EMPLOYEES，查看错误提示，如实例 15-39 所示。

【实例 15-39】查询表空间 LIN 中的表 EMPLOYEES。

```
SQL> show user
USER 为"SYS"
SQL> select *
  2  from scott.employees;
from scott.employees
          *
ERROR 位于第 2 行:
ORA-00376: 此时无法读取文件 11
ORA-01110: 数据文件 11: 'D:\TEMP\LIN.DBF'
```

在上例中，我们先查看了当前用户为 SYS，所以在查询表 EMPOYEES 时必须指明该表所属的用户 SCOTT。查询结果说明，无法读取文件 11，此时的 11 为文件号，并且 Oracle 知道无法读取的数据文件的位置，遇到这种情况，用户可以根据文件信息，查找该文件的状态，很容易确

定问题在哪。

2. 只读管理

只读管理就是把表空间置为只读状态，这样的表空间中的数据只能被用户读取，而不能做任何修改或插入操作，在数据库设计时，如果有的数据是静态数据，则可以将存储这些数据的表放在一个表空间中，只读管理的表空间不被重做日志保护，减少重做日志文件的大小。

把一个表空间置为只读状态的指令为 ALTER DATABASE tablespace_name READ ONLY。下面通过实例 15-40 演示如何将 USERS 表空间设置为只读状态。

【实例 15-40】将 USERS 表空间设置为只读状态。

```
SQL> alter tablespace users read only;

表空间已更改。
```

再通过实例 15-41 查看该表空间的状态。

【实例 15-41】查看表空间的状态。

```
SQL> select tablespace name,status
  2  from dba_tablespaces;

TABLESPACE NAME      STATUS
-------------------------------------- --------------
SYSTEM               ONLINE
UNDOTBS1             ONLINE
SYSAUX               ONLINE
TEMP                 ONLINE
USERS                READ ONLY
EXAMPLE              ONLINE

已选择 6 行。
```

我们看到该表空间 USERS 的状态 STATUS 为 READ ONLY，已经设置为只读状态。为了验证只读表空间的只读性，通过一系列的实例来验证，首先通过实例 15-42 查看表空间 USERS 中 SCOTT 用户的表信息。

【实例 15-42】查看表空间 USERS 中 SCOTT 用户的表信息。

```
SQL> select owner,tablespace_name,table_name
  2  from dba_tables
  3* where owner = 'SCOTT'

OWNER                TABLESPACE_NAME      TABLE_NAME
---------------------------------- -------------------------------------
SCOTT                USERS                DEPT
SCOTT                USERS                EMP
SCOTT                USERS                BONUS
SCOTT                USERS                SALGRADE
```

在 USERS 表空间中有 SCOTT 用户的 4 个表，分别是 DEPT、EMP、BONUS 和 SALGRADE，下面查询表 DEPT 的信息，如实例 15-43 所示。

【实例 15-43】查询表 DEPT 的信息。

```
SQL> select *
  2  from scott.dept;

    DEPTNO  DNAME            LOC
---------------------------------------------------- ----------------
        10  ACCOUNTING       NEW YORK
        20  RESEARCH          DALLAS
        30  SALES             CHICAGO
        40  OPERATIONS        BOSTON
```

显然可以查询该表空间 USERS 中表内的数据，下面再修改表空间 USERS 中表 DEPT 内的数据，删除 DEPTNO 为 40 的记录，看是否成功，如实例 15-44 所示。

【实例 15-44】删除表 DEPT 中 DEPTNO 为 40 的记录。

```
SQL> delete from scott.dept
  2  where deptno = 40;
delete from scott.dept
                  *
第 1 行出现错误:
ORA-00372: 此时无法修改文件 4
ORA-01110: 数据文件 4: 'D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF'
```

我们看到，无法删除表空间 USERS 中表 DEPT 内的记录，因为该表空间设置为只读状态，该表空间中的数据是无法做任何更改的。

在需要的时候，可以将一个设置为只读状态的表空间设置为正常状态，即可读、可写状态，如实例 15-45 所示。

【实例 15-45】将设置为只读状态的表空间 USERS 设置为正常状态。

```
SQL> alter tablespace users read write;

表空间已更改。
```

通过实例 15-46 验证是否将表空间 USERS 设置为正常状态。

【实例 15-46】验证是否将表空间 USERS 设置为正常状态。

```
SQL> select tablespace_name,status
  2  from dba_tablespaces;

TABLESPACE_NAME      STATUS
------------------------------------ ---------------
SYSTEM               ONLINE
UNDOTBS1             ONLINE
SYSAUX               ONLINE
```

```
TEMP                    ONLINE
USERS                   ONLINE
EXAMPLE                 ONLINE

已选择 6 行。
```

上述输出说明表空间 USERS 的状态为 ONLINE，说明已经将表空间 USERS 设置为可读、可写的正常状态。

15.5.2 表空间的内容管理

表空间的内容管理涉及修改表空间的大小、删除表空间以及修改表空间的存储参数等。

1. 修改表空间的大小

修改表空间的大小有 4 种方法。

- 在创建表空间时，使用 AUTOEXTEND ON 子句使得表空间在需要时可以自动扩展。
- 在创建表空间后使用 ALTER DATABASE DATAFILE file_name AUTOEXTEND ON 修改不能自动扩展的表空间的数据文件。
- 在表空间中增加数据文件。
- 修改数据文件的大小，即重新设置表空间中某个数据文件的大小。

下面通过实例依次演示上述 4 种修改表空间的方法。

首先演示在创建表空间时，使用 AUTOEXTEND ON 子句创建可以自动扩展的表空间，如实例 15-47 所示。

【实例 15-47】创建数据文件自动扩展的表空间。

```
SQL> create tablespace manager_tbs1
  2  datafile 'd:/tbs_manager1/tbs1.dbf'
  3  size 100M
  4  autoextend on;

表空间已创建。
```

在创建表空间 MANAGER_TBS1 时，在 DATAFILE 子句后使用了 SIZE 指定该数据文件的大小。使用 AUTOEXTEND 子句指定该数据文件为自动扩展，如实例 15-48 所示。

【实例 15-48】查看表空间 MANAGER_TBS1 的数据文件的扩展方式。

```
SQL> select file_name,tablespace_name,blocks,status,autoextensible
  2  from dba_data_files
  3* where tablespace_name like 'MAN%'

FILE_NAME                     TABLESPACE_NAME  BLOCKS STATUS   AUT
----------------------------------------------- ----------------------------
D:\TBS_MANAGER1\TBS1.DBF   MANAGER_TBS1      12800 AVAILABLE YES
```

在创建表空间后，使用 ALTER DATABASE 命令修改表空间中的数据文件为自动扩展，如实例 15-49 所示，先创建一个表空间 MANAGE_TBS，再查看该表空间是否是可自动扩展的。

【实例 15-49】创建一个表空间 MANAGE_TBS。

```
SQL> create tablespace manage_tbs
  2  datafile 'd:\tbs_manager\tbs01.dbf'
  3  size 50M
  4  uniform size 1M;

表空间已创建。

SQL> select tablespace name,file name,autoextensible
  2  from dba_data_files;

TABLESPACE NAME      FILE NAME                                         AUT

-------------------- ------------------------------------------------
USERS      D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF       YES
SYSAUX    D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF      YES
UNDOTBS1  D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF    YES
SYSTEM    D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF      YES
EXAMPLE  D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF     YES
MANAGE_TBS  D:\TBS_MANAGER\TBS01.DBF                              NO
已选择 6 行。
```

在实例 15-49 中，先创建了一个表空间 MANAGE_TBS，然后查看了该表空间的 AUTOEXTENSIBLE 属性值为 NO，也就是该表空间在空间不足时不能自动扩展，显然这样的表空间缺少灵活性，也对 DBA 维护数据库增加了负担，现在使用 ALTER DATABASE 指令修改该表空间中的数据文件为自动扩展，且需要空间时自动扩展 1M 空间，如实例 15-50 所示。

【实例 15-50】使用 ALTER DATABASE 命令修改表空间中的数据文件为自动扩展。

```
SQL> alter database datafile
  2  'd:\tbs_manager\tbs01.dbf' autoextend on
  3  next 1M;

数据库已更改。
```

为了确认修改结果，使用实例 15-51 验证表空间 MANAGE_TBS 的数据文件是否处于自动扩展模式。

【实例 15-51】查询表空间 MANAGE_TBS 的数据文件的自动扩展模式。

```
SQL> col file name for a30
SQL> select tablespace_name,file_name,autoextensible
  2  from dba_data_files
  3* where tablespace name like 'MAN%'

TABLESPACE_NAME      FILE_NAME                      AUT
-------------------- ------------------------------------------------
```

```
MANAGE_TBS          D:\TBS_MANAGER\TBS01.DBF      YES
```

我们看到，表空间 MANAGE_TBS 中有一个数据文件，该数据文件的自动扩展属性值为 YES，所以该文件在表空间不足时，可以自动扩展，每次自动扩展的空间大小为 1M。

前面使用 ALTER DATABASE 命令修改了表空间中的数据文件为可以自动扩展，现在用另一种方法，即在表空间中增加一个数据文件的方式增加表空间容量。如实例 15-52 所示，向表空间 MANAGE_TBS 中添加一个数据文件。



在实例 15-52 中，我们假设表空间 MANAGE_TBS 中的数据文件没有设置为自动扩展。

【实例 15-52】向表空间 MANAGE_TBS 中增加了一个数据文件。

```
SQL> alter tablespace manage tbs
  2  add datafile 'D:\TBS_MANAGER\TBS02.DBF'
  3  size 50M;

表空间已更改。
```

此时，已成功向表空间 MANAGE_TBS 中增加了一个数据文件，该文件大小为 50M，这样通过增加文件的方式增加了表空间的容量，通过实例 15-53 验证表空间 MANAGE_TBS 中的数据文件信息。

【实例 15-53】验证表空间 MANAGE_TBS 中的数据文件信息。

```
SQL> select tablespace_name,file_name,status,autoextensible
  2  from dba_data_files
  3  where tablespace_name = 'MANAGE_TBS';

TABLESPACE NAME     FILE NAME                          STATUS    AUT
-------------------- ------------------------------------------------
MANAGE_TBS          D:\TBS_MANAGER\TBS01.DBF      AVAILABLE  NO
MANAGE_TBS          D:\TBS_MANAGER\TBS02.DBF      AVAILABLE  NO
```

通过实例 15-53 可以看到表空间 MANAGE_TBS 中有两个文件表空间，MANAGE_TBS 的大小是两个数据文件大小之和。

不论使用哪种方法修改表空间的大小，都不能超过数据文件所在的磁盘空间，所以要定期查看告警日志文件，查看是否有表空间不足的警告，并及时处理，否则会造成数据挂起或者数据库关闭。

下面将表空间 MANAGE_TBS 中的数据文件 D:\TBS_MANAGER/TBS01.DBF 修改为 100M（修改前为 50M），如实例 15-54 所示。

【实例 15-54】修改表空间 MANAGE_TBS 中的数据文件。

```
SQL>conn system/oracle
已连接。
```

```
SQL> alter database
  2  datafile 'D:\TBS_MANAGER\TBS01.DBF' resize 100M;

数据库已更改。
```

使用 ALTER DATABASE 指令修改了表空间 MANAGE_TBS 中的一个数据文件，在修改前必须查询该表空间中的数据文件，而后再使用 ALTER DATABASE 指令修改，因为在修改时不会有任何表空间的提示。

在修改了数据文件 D:\TBS_MANAGER/TBS01.DBF 为 100M 后，再用实例 15-55 验证修改结果。

【实例 15-55】验证在实例 15-54 中修改的数据文件大小。

```
SQL> select tablespace_name,file_name,bytes,status
  2  from dba_data_files
  3* where tablespace_name = 'MANAGE_TBS'

TABLESPACE_NAME   FILE_NAME                    BYTES STATUS
------------------------------ --------------------
MANAGE_TBS        D:\TBS_MANAGER\TBS01.DBF  104857600 AVAILABLE
```

输出中 BYTES 为 104857600 字节（100M），说明已经成功修改了表空间中的数据文件的大小，即增大了表空间的容量，通过间接的方式修改了表空间的容量。

2. 修改表空间的存储参数

修改表空间的存储参数只对数据字典管理的表空间有效，在 Oracle 10g 和 9i 中创建的表空间默认都为本地管理的表空间，为了演示如何修改表空间的存储参数，我们使用实例 15-47 修改在前面创建的数据字典管理的表空间 tianjin_data 的存储参数 MINIMUM EXTENT，即修改该表空间分配的最小 EXTENT 尺寸，修改为 2M，如实例 15-56 所示。

【实例 15-56】修改该表空间分配的最小 EXTENT 尺寸。

```
SQL> alter tablespace tianjin_data
  2  minimum extent 2M;
表空间已更改。
```

【实例 15-57】修改表空间 tianjin_data 的默认存储子句。

```
SQL> alter tablespace tianjin_data
  2  default storage(initial 2M next 2M maxextents 50);
表空间已更改。
```

为了验证修改结果，我们使用实例 15-58 查看修改后的表空间 tianjin_data 的存储参数。

【实例 15-58】查看修改后的表空间 tianjin_data 的存储参数。

```
SQL> set line 100
SQL>select tablespace_name, initial_extent,next_extent,max_extents,
min_extlen
  2  from dba_tablespaces
```

```
  3* where tablespace_name = 'TIANJIN_DATA'
TABLESPACE_NAME   INITIAL_EXTENT NEXT_EXTEN MAX_EXTENTS MIN_EXTLEN
-------------------------------- ------------------ -------------------- --------
TIANJIN_DATA           2097152        2097152         50          2097152
```

上述输出结果表明已成功修改了表空间 tianjin_data 的存储参数。在修改参数时，我们使用 M 作为单位，而输出中默认使用字节。



在 Oracle 10g 中已经不允许创建数据字典管理的表空间。只有在 Oracle 9i 以及之前的版本才可以创建数据字典管理的表空间。

3. 删除表空间

当不需要一个表空间时，可以删除该表空间以释放磁盘空间，删除表空间的语法格式比较简单，如下所示：

```
DROP TABLESPACE tablespace_name
[INCLUDING CONTENTS [AND DATAFILES] [CASCADE CONSTRAINTS]]
```

各参数的含义如下。

- Tablespace_name：要删除的表空间名。
- INCLUDING CONTENTS：删除表空间中的所有区段（EXTENTS）。
- AND DATAFILES：删除表空间中的数据文件，该文件是一个 Oracle 格式的操作系统文件。
- CASCADE CONSTRAINTS：删除和该表空间中的表相关的引用完整性约束，外部表会引用该表空间中的表的唯一键等作为外部表的引用。

下面给出一个实例，删除表空间 MANAGE_TBS 并且删除该表空间中的一个数据文件，如实例 15-59 所示。

【实例 15-59】删除表空间并且删除该表空间中的数据文件。

```
SQL> drop tablespace manage_tbs including contents and datafiles;

表空间已删除。
```

此时，和该表空间相关联的所有数据文件都一同删除。读者可以自己验证这个结果。



如果使用 drop tablespace manage_tbs 指令删除该表空间，则只是删除了控制文件中指向该数据文件的文件指针，而实际的数据文件不会被删除。

在删除表空间后，该数据库中不再有该表空间的任何数据，数据库不再管理这些数据文件，只读状态的表空间和表空间的区段仍然可以顺利删除，在删除表空间时，Oracle 推荐将表空间置于脱机状态，以避免仍然有事务在操作表空间中的区段。

15.6　数据文件的管理

数据文件逻辑地存放在表空间中，管理数据文件涉及修改数据文件的大小、迁移数据文件等。这里主要讲解迁移数据文件的知识。

迁移数据文件是指把当前表空间中的数据文件迁移到其他磁盘空间，可以想象在生产数据库中，当一个表空间所在的磁盘满时，为了使数据库系统正常运行，必须将其中的数据文件迁移到其他空闲磁盘。

迁移数据文件主要分为两种，即迁移系统表空间中的文件和迁移非系统表空间中的文件，下面分别进行介绍。

1. 迁移系统表空间中的数据文件

先查看当前系统表空间中的数据文件信息，如下所示：

```
SQL> select tablespace_name,file_name,bytes,status
  2  from dba_data_files
  3  where tablespace_name = 'SYSTEM';

TABLESPACE_NAME      FILE_NAME                          BYTES STATUS
-------------------- ----------------------------------------------
SYSTEM              F:\ORACLE\PRODUCT\10.2.0\ORADA  503316480 AVAILABLE
                   TA\ORCL\SYSTEM01.DBF
```

目的是确定系统表空间 SYSTEM 中的数据文件位置，FILE_NAME 属性值给出了文件的具体路径和文件名，即 F:\oracle\product\10.2.0\oradata\orcl\system01.dbf。

使用具有 DBA 权限的用户登录数据库服务器，然后关闭数据库：

```
SQL> conn system/oracle as sysdba
已连接。
SQL> shutdown immediate;
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
```

拷贝 SYSTEM 表空间的数据文件到新的目录下，该新目录为 D:\SYSTEM。此时既可以使用操作系统指令拷贝，也可以使用 SQL*Plus 的 HOST COPY，在 UNIX 系统中为 HOST CP。

```
SQL> host copy F:\oracle\product\10.2.0\oradata\orcl\system01.dbf d:\system
已复制          1 个文件。
```

因为系统表空间 SYSTEM 中的数据文件大小不同，所以需要等待的时间也有差异，如上述输出所示，成功拷贝了 SYSTEM 表空间中的文件。

打开数据库到 MOUNT 状态，然后使用 ALTER DATABASE RENAME FILE 指令迁移数据文件，这里的作用是告诉数据库做了哪些数据迁移和数据迁移地点。数据库再次打开数据文件时，会知道到哪里打开数据文件。

```
SQL> startup mount;
ORACLE 例程已经启动。

Total System Global Area  603979776 bytes
Fixed Size                  1250380 bytes
Variable Size             176163764 bytes
Database Buffers          419430400 bytes
Redo Buffers                7135232 bytes
数据库装载完毕。
SQL> alter database
  2  rename file 'F:\oracle\product\10.2.0\oradata\orcl\system01.dbf'
  3  to 'd:\system\system01.dbf';

数据库已更改。
```

当打开数据库到 MOUNT 状态时启动实例，但是 Oracle 不会打开数据文件，所以迁移数据文件的指令可以成功执行。

打开数据库，使用指令 ALTER DATABASE OPEN 实现：

```
SQL> alter database open;

数据库已更改。
```

在 Oracle 10g 中，需要先执行介质回复，否则会提示错误，要求介质回复。介质回复的指令为 RECOVER DATABASE，然后使用 ALTER DATABASE OPEN 指令打开数据库。

在迁移了 SYSTEM 表空间的数据文件后，通过实例 15-60 验证修改结果。

【实例 15-60】验证迁移后的系统表空间。

```
SQL> col file_name for a50
SQL> run
  1  select tablespace_name,file_name,status
  2  from dba_data_files
  3* where tablespace_name like 'SYS%'

TABLESPACE_NAME      FILE_NAME                                          STATUS
-------------------- --------------------------------------------------
SYSAUX      F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF AVAILABLE
SYSTEM      D:\SYSTEM\SYSTEM01.DBF                            AVAILABLE
```

此时，系统表空间的数据文件已经迁移到新的磁盘目录下。

2. 迁移非系统表空间

在 Oracle 中，要求这种非系统表空间没有活跃的还原段、临时段、排序段等，这种非系统表空间才可以迁移。下面迁移数据库中的 MANAGE_TBS，该表空间中有一个数据文件 D:\TBS_MANAGER\TBS01.DBF，下面将它迁移到 F:\TBS_MANAGER 磁盘目录下。

把表空间 MANAGE_TBS 设置为脱机：

```
SQL> alter tablespace manage_tbs offline;

表空间已更改。
```

拷贝数据文件到新的磁盘目录下，文件名不变：

```
SQL> host copy d:\tbs_manager\tbs01.dbf f:\tbs_manager
已复制         1 个文件。
```

使用指令 ALTER DATABASE RENAME DATAFILE 迁移数据文件：

```
SQL> alter tablespace manage_tbs
  2  rename datafile 'd:\tbs manager\tbs01.dbf'
  3* to 'f:\tbs_manager\tbs01.dbf'

表空间已更改。
```

把表空间联机：

```
SQL> alter tablespace manage_tbs online;

表空间已更改。
```

经过上述步骤，已经迁移了非系统表空间 MANAGE_TBS 中的数据文件到新的磁盘目录下，下面再通过实例 15-61 验证修改结果。

【实例 15-61】验证对非系统表空间的迁移。

```
SQL> select tablespace_name,file_name,status
  2  from dba data files
  3* where tablespace_name like 'MAN%'

TABLESPACE_NAME      FILE_NAME                          STATUS
-------------------- ------------------------------------------------
MANAGE_TBS          F:\TBS_MANAGER\TBS01.DBF        AVAILABLE
MANAGER_TBS1        D:\TBS_MANAGER1\TBS1.DBF        AVAILABLE
```

表空间 MANAGE_TBS 中的数据文件名为 F:\TBS_MANAGER\TBS01.DBF，显然已成功迁移了数据字典，把表空间 MANAGE_TBS 中的数据文件迁移到了 F 盘的目录下。

15.7　本章小结

本章是内容较为丰富的一章，表空间和数据文件是 Oracle 数据库中非常重要的两个概念，表空间是一个逻辑概念，它和段、区段和数据库块组成了数据库的逻辑结构，而数据文件和操作系统块组成了数据库的物理结构，采用逻辑结构和物理结构的结构模式是 Oracle 为了满足其在不同操作系统之间能够方便地移植而设计的。

本章讲解了表空间的逻辑结构和物理结构之间的关系，从而理解 Oracle 如何操作数据文件以及操作系统如何管理和操作数据文件。Oracle 把表空间的管理方式分为数据字典管理的表空间

和本地管理的表空间，而本地管理的表空间是 Oracle 推荐的方式，在 Oracle10g 和 Oracle 11g 中是默认的表空间管理方式。表空间的维护是 DBA 的一项重要任务，如还原表空间、临时表空间和默认临时表空间、创建大文件表空间等。

表空间创建后，可以通过指令修改表空间的参数，但是只对数据字典管理的表空间有效，用户也可以把表空间中的数据文件迁移到其他磁盘，以减少当前数据文件所在磁盘的压力，在不需要表空间时，可以采用不同的方式删除表空间，删除表空间是需要谨慎对待的，因为一旦删除数据很难恢复。

第 16 章

重做日志管理

重做日志是 Oracle 数据库中很重要的一部分内容，在数据库恢复时，往往需要对重做日志的工作原理清晰理解，以及学会如何配置和维护重做日志。本章首先介绍了 Oracle 引入重做日志的原因，介绍重做日志的工作过程，进而介绍 Oracle 的重做日志结构、重做日志组和重做日志成员。日志切换和检查点事件是涉及重做日志维护的重要事件，通过对这两个事件的学习有助于实现重做日志的维护。最后介绍与重做日志相关的报警文件信息和 LGWR 追踪文件，这对于故障处理提供了很好的线索，最后介绍归档重做日志。

16.1 引入重做日志的原因

可以用 4 个字来说明 Oracle 引入重做日志的原因：数据恢复。在数据库运行过程中，用户更改的数据会暂时存放在数据库高速缓冲区中，而为了提高写数据库的速度，不是一旦有数据变化，就把变化的数据写到数据文件中，频繁的读写磁盘文件使得数据库系统效率降低，所以，要等到数据库高速缓冲区中的数据达到一定的量或者满足一定条件时，DBWR 进程才会将变化了的数据提交到数据库中，也就是 DBWR 将变化了的数据写到数据文件中。在这种情况下，如果在 DBWR 把变化了的更改写到数据文件之前发生了宕机，那么数据库高速缓冲区中的数据就会全部丢失，如果在数据库重新启动后无法恢复这部分用户更改的数据，显然是不合适的。

而重做日志就是把用户变化了的数据首先保存起来，其中 LGWR 进程负责把用户更改的数据先写到重做日志文件中。在数据库原理课程中，这种机制也叫做日志写优先。这样在数据库重新启动时，数据库系统会从重做日志文件中读取这些变化了的数据，将用户更改的数据提交到数据库中，并写入数据文件。

为了提高磁盘效率、防止重做日志文件的损坏，Oracle 引入了一种重做日志结构，如图 16-1 所示。

在该示例图中，重做日志文件结构由三个重做日志组组成，每个重做日志组中有两个重做日志成员（重做日志文件），当然可以有更多的重做日志组，每个组中也可以有更多的重做日志成员。数据库系统会先使用重做日志组 1，但该组写满后，就切换到重做日志组 2，再写满后，继续切换到重做日志组 3，然后循环使用重做日志组 1，Oracle 以这样的循环方式使用重做日志组。

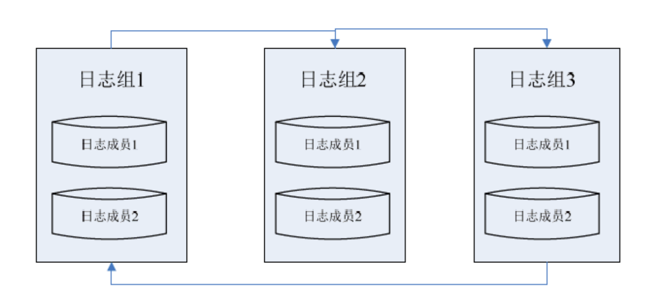


图 16-1　重做日志文件结构

该结构直观地说明了重做日志文件的组成，Oracle 规定每个数据库实例至少有两个重做日志组，每个重做日志组至少有一个重做日志文件。当重做日志组中有多个日志成员时，每个重做日志成员的内容相同，Oracle 会同步同一个重做日志组中的每个成员。在工作过程中，Oracle 是循环的使用重做日志组，当一个重做日志组写满时，就自动进行日志切换，切换到它可以找到的其他重做日志组，并为该日志组设置一个日志序列号。在必要的条件下也可以实现强制日志切换。

如果没有启动归档日志，当一个循环结束，再次使用先前的重做日志组时，会以覆盖的方式向该组的重做日志文件中写数据。在非归档模式下，重新使用新的联机重做日志前，DBWR 进程需要将所有的数据更改写到数据文件中，有时也称为 DBWR 归档，所以，对于生产数据库要求工作在归档模式下。

如果数据库处于归档模式下，当前正在使用的重做日志写满后，Oracle 会关闭当前的日志文件，ARCH 进程把旧的重做日志文件中的数据移动到归档重做日志文件中。归档完成后，寻找下一个可用重做日志组，找到该组中可用的日志文件，打开该文件并实现写操作。归档进程并不是一直存在。



如果数据库处于归档模式，在归档进程 ARCH 把联机重做日志移动到归档日志前，Oracle 无法使用一个已经关闭的重做日志，即如果 ARCH 没有完成，就没有已经归档的联机重做日志可以用于切换，只有 ARCH 释放了联机重做日志后，数据库才可以继续工作。

16.2　获取重做日志文件信息

在第 16.1 小节中介绍了 Oracle 为何引入重做日志文件，以及重做日志文件的工作机制和文件结构，那么在一个数据库系统中如何查看关于重做日志文件的信息呢？我们通过两个动态数据字典视图 v$log 和 v$logfile 来获取关于重做日志文件的信息。

16.2.1　v$log 视图

数据字典视图 v$log 记录了当前数据库的日志组号、日志序列号、每个日志文件的大小（以

字节为单位）、每个日志组的成员数量以及日志组的当前状态。实例 16-1 说明了使用 v$log 查看重做日志信息的方法。

【实例 16-1】使用 v$log 查看重做日志信息。

```
SQL>conn /as sysdba
SQL> select group#,sequence#,bytes,members,archived,status
  2  from v$log;

    GROUP#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- --- ------------
         1         39   10485760          1 NO  CURRENT
         2         38   10485760          1 NO  INACTIVE
         3         37   10485760          1 NO  INACTIVE
```

该实例 16-1 的输出说明，当前有三个日志组，与每个日志文件对应的日志序列号是全局唯一的，同一个日志组中的日志序列号相同，用户数据库恢复时使用每个日志组的成员数量及每个日志组的当前状态。重做日志组 1 为当前正在使用的重做日志组，该日志组中有最大日志序列号，该日志文件还没有归档。

16.2.2 v$logfile 视图

数据字典视图 v$logfile 记录了当前日志组号、该日志组的状态、类型和日志组成员信息，下面用实例 16-2 说明该数据字典视图的输出。

【实例 16-2】使用数据字典视图 v$logfile 查看重做日志组信息。

```
SQL>conn /as sysdba
SQL> col member for a50
SQL> select group#,status,type,member
  2  from v$logfile;

    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- ------------------------------------
         3 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
         2 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
         1         ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
```

在解释输出结果前，先介绍一下 STATUS 参数的含义。

- STALE：说明该文件内容为不完整的。
- 空白：说明该日志组正在使用。
- INVALID：表示该文件不能被访问。
- DELETED：表示该文件已经不再使用。

下面继续说明实例 16-2 的输出，该数据库系统有 3 个重做日志组，每个日志组有一个重做日志成员，且都为联机（ONLINE）重做日志文件。其实 DBA 如果看到这样的情况，应该知道需要增加重做日志成员，并且把每个日志组的重做日志成员分布在不同磁盘上。

16.3 重做日志组

Oracle 要求最少两个重做日志组，每个日志组至少有一个日志成员，而在生产数据库中至少需要 3 个重做日志组，而每个重做日志组需要多于 3 个重做日志成员，这些日志成员最好部署在不同磁盘的不同目录下，由于重做日志文件在数据库恢复中的重要性，分布式部署的目的就是为了防止磁盘损坏造成的重做日志失效。

16.3.1 添加重做日志组

向当前数据库中添加一个新的日志组的语法格式为：

```
ALTER DATABASE [database_name]
        ADD LOGFILE [GROUP number] filename SIZE n
        [,ADD LOGFILE [GROUP number]   filename SIZE n……]
```



filename 为日志组成员的文件目录和文件名称。参数 GROUP number 可以不用，Oracle 会自动生成一个日志组号，该日志组号在原有日志组号的基础上加 1。

通过实例 16-3 演示如何添加一个重做日志组。

【实例 16-3】添加一个重做日志组。

```
SQL> ALTER DATABASE ADD LOGFILE GROUP 4
  2  ('d:\temp\redo04a.log',
  3   'd:\temp\redo04b.log')
  4  SIZE 10M;

数据库已更改。
```

在实例 16-3 中，向当前数据库添加一个重做日志组，日志组号为 4，如果不选择 GROUP 参数，则默认在原有重做日志组号的基础上自动增长，如原来最大的日志组号为 2，则此时新建的默认组号为 3，依次类推。在日志组 4 中，有两个日志成员，大小都为 10M。



在实例 16-3 中，日志成员的目录必须存在，即 d:\temp 目录必须存在。

为了验证添加日志组的结果，可使用实例 16-4 进行验证。

【实例 16-4】验证添加日志组的结果。

```
SQL> select *
  2  from v$logfile;

GROUP# STATUS  TYPE    MEMBER
```

```
------------ -------------- ----------- ----------------------
      3 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
      2 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
      1         ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
      4         ONLINE  D:\TEMP\REDO04A.LOG
      4         ONLINE  D:\TEMP\REDO04B.LOG
```

实例 10-4 的输出结果说明，已成功添加重做日志组，该日志组有两个日志成员，即为 D:\TEMP\REDO04A.LOG 和 D:\TEMP\REDO04B.LOG，这两个重做日志成员都处于联机状态。

下面再添加一个重做日志组，此时不选择 GROUP 参数，向该日志组中添加三个日志成员，成员大小都为 10M，如实例 16-5 所示。

【实例 16-5】添加一个重做日志组并向该日志组中添加三个日志成员。

```
SQL>  ALTER DATABASE ADD LOGFILE
  2   ('d:\disk6\redo05a.log',
  3    'd:\disk6\redo05b.log',
  4    'd:\disk6\redo05c.log')
  5  SIZE 10 M;

数据库已更改。
```

当前的重做日志组有 4 个，实例 16-5 没有指定 GROUP 号，此时 Oracle 会为这个重做日志组自动生成一个组号，即在原有的日志组号的基础上加 1，所以新建的日志组号应该为 5。

【实例 16-6】验证实例 16-5 的执行结果。

```
SQL>conn /as sysdba
SQL> select *
  2  from v$logfile;

GROUP# STATUS  TYPE    MEMBER
----------- ------------- ------------- -------
      3 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
      2 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
      1         ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
      4         ONLINE  D:\TEMP\REDO04A.LOG
      4         ONLINE  D:\TEMP\REDO04B.LOG
      5         ONLINE  D:\DISK6\REDO05A.LOG
      5         ONLINE  D:\DISK6\REDO05B.LOG
      5         ONLINE  D:\DISK6\REDO05C.LOG
```

显然，我们新添加的重做日志组号为 5，该日志组共有 3 个重做日志文件，通过实例 16-7 查询当前重做日志组的使用情况，查看新建的重做日志组的状态。

【实例 16-7】查询当前重做日志组的使用情况。

```
SQL> select group#,sequence#,bytes,members,archived,status
  2  from v$log;

   GROUP#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
```

```
---------- ---------- ---------- ---------- --- ----------
         1         39   10485760          1 NO  CURRENT
         2         38   10485760          1 YES INACTIVE
         3         37   10485760          1 YES INACTIVE
         4          0   10485760          2 YES UNUSED
         5          0   10485760          3 YES UNUSED
```

重做日志组 4 和重做日志组 5 是新建的重做日志组，二者的状态都为 UNSUED 未使用，重做日志组 4 有 2 个日志成员，每个成员的大小为 10485760 个字节，重做日志组 5 有 3 个日志成员，每个成员的大小为 10485760 个字节。因为两个重做日志组还没有使用，所以 Oracle 没有分配日志序列号。

16.3.2 删除重做日志组

在不需要一个重做日志组时，可以删除该日志组。删除重做日志组的语法格式为：

```
ALTER DATABASE [database_name]
DROP LOGFILE {GROUP n|('filename'[,'filename']…)}
    [{GROUP n|('filename'[,'filename']…)}]…
```

在上述语法中符号“|”表示“或”的关系，而符号“[]”表示可选。通过实例 16-8 来演示如何删除一个不用的重做日志组。

【实例 16-8】删除重做日志组。

```
SQL> alter database drop logfile group 4,group 5;

数据库已更改。
```



当前的重做日志组和处于 ACTIVE 状态的重做日志组都无法删除，如果要删除当前在用的日志组，必须先进行日志切换。在删除一个日志组时可以使用 GROUP 参数直接删除该日志组，也可以指定删除该日志组中的所有重做日志文件来达到删除日志组的效果。读者可以自己尝试这种方法。

使用实例 16-9 和实例 16-10 查询日志组 5 是否删除。

【实例 16-9】验证日志组 5 的成员是否删除。

```
SQL>conn /as sysdba
SQL> select *
  2  from v$logfile;

    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- -----------------------------------
         3 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
         2 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
         1         ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
```

【实例 16-10】验证日志组 5 是否删除。

```
SQL> select group#,sequence#,bytes,members,archived,status
  2  from v$log;

    GROUP#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- --- ----------
         1         39   10485760          1 NO  CURRENT
         2         38   10485760          1 YES INACTIVE
         3         37   10485760          1 YES INACTIVE
```

实例 16-9 和实例 16-10 都说明数据库系统已经将重做日志组 4 和重做日志组 5 删除。但是需要读者注意，使用指令删除重做日志组会留下垃圾文件，也就是说在删除了重做日志组后，作为重做日志组成员的操作系统文件还存在，如图 16-2 所示。

所以，必须使用操作系统指令手工删除这些垃圾文件。

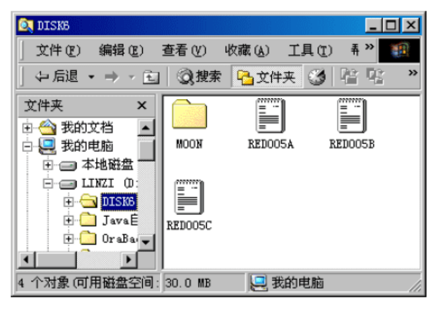


图 16-2　删除重做日志组后的垃圾文件

16.4　重做日志成员

重做日志成员是与重做日志组相对应的一个概念，在一个重做日志组中允许有一个重做日志成员，但是在生产数据库中要求多于 4 个日志文件，并且同一个日志组中的多个重做日志文件分布在不同磁盘的不同目录下。在同一个日志组中，日志成员的大小相同。本节将讲解如何添加和删除重做日志成员，以及重做日志成员的维护。

16.4.1　添加重做日志成员

在每个重做日志组中至少要有一个日志成员，但是为了防止单点失效的发生，最好多设置几个重做日志成员，并存储在不同的磁盘空间中。这样的冗余设置可以极大地提高重做日志文件的可靠性。

向一个重做日志组中添加日志成员的语法格式如下所示：

```
ALTER DATABASE [databasename]
ADD LOGFILE MEMBER
        ['filename' [REUSE]
            [,'filename'[REUSE]]……
TO {GROUP n
    |('filename'[,'filename']……)}
]……
```

下面通过实例 16-11 来演示如何向重做日志组中添加重做日志成员，此时无论是否是当前正在使用的重做日志组，都可以添加重做日志成员，该实例中分别向重做日志 1、2、3 添加一个重做日志成员。

【实例 16-11】向重做日志 1、2、3 添加一个重做日志成员。

```
SQL> alter database add logfile member
  2  'd:\temp\redo01a.log' to group 1,
  3  'd:\temp\redo02a.log' to group 2,
  4  'd:\temp\redo03a.log' to group 3;

数据库已更改。
```

在成功添加后，用实例 16-12 和实例 16-13 验证添加结果。

【实例 16-12】验证日志组的成员数。

```
SQL> select group#,sequence#,bytes,members,archived,status
  2  from v$log;

    GROUP#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- --- ------------
         1         39   10485760          2 NO  CURRENT
         2         38   10485760          2 YES INACTIVE
         3         37   10485760          2 YES INACTIVE
         4          0   10485760          2 YES UNUSED
```

在上述输出中，重做日志组 1、2 和 3 的 MEMBERS 都为 2，说明这些重做日志组有 2 个重做日志成员。再通过实例 16-13 验证添加的重做日志文件信息。

【实例 16-13】验证添加的重做日志组以及对应的成员信息。

```
SQL> select *
  2  from v$logfile
  3  order by group#;

    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- -----------------------------------
         1         ONLINE  F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
         1 INVALID ONLINE  D:\TEMP\REDO01A.LOG
         2 STALE   ONLINE  F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
         2 INVALID ONLINE  D:\TEMP\REDO02A.LOG
         3 STALE   ONLINE  F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
         3 INVALID ONLINE  D:\TEMP\REDO03A.LOG
         4         ONLINE  D:\TEMP\REDO04A.LOG
         4         ONLINE  D:\TEMP\REDO04B.LOG
```

实例 16-13 中，使用了 order by 子句对输出进行排序，这样就可方便查看每一个重做日志组的成员。在上述输出中，重做日志组 1、2 和 3 都新增了一个重做日志成员。



如果添加的日志成员文件已经存在，则需要使用 REUSE 参数，并且日志成员要用全目录格式，不要使用相对目录的形式，否则，Oracle 数据库服务器会在默认路径下建立该重做日志文件。

16.4.2　删除重做日志成员

如果不需要一个重做日志成员，可以删除掉，通常我们所重做的日志维护就是删除和重建重做日志的过程，对于一个损坏的重做日志，即使没有发现日志切换时无法成功，数据库最终也会挂起，当然如果读者对于重做日志成员做了很好地分布存储，出现这种情况的可能性很小。但是，一旦出现重做日志文件受损的情况就要及时修复，也就是删除掉该文件，然后重建。

删除重做日志文件的语法格式为：

```
ALTER DATABASE [database_name]
DROP LOGFILE MEMBER 'filename' [,'filename']……
```

下面用实例 16-14 演示如何删除一个重做日志成员。此时，只需要知道重做日志成员的目录和文件名，而不必知道该日志成员所属的日志组。在该实例中，删除重做日志组 4 中的一个日志成员。

【实例 16-14】删除重做日志组中的一个日志成员。

```
SQL> alter database drop logfile member 'D:\TEMP\REDO04A.LOG';

数据库已更改。
```

通过实例 16-15 查询重做日志组 4 的日志成员信息。

【实例 16-15】查询重做日志组 4 的日志成员信息。

```
SQL> select *
  2  from v$logfile
  3  where group# = 4;

    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- --------------------
         4         ONLINE  D:\TEMP\REDO04B.LOG
```

输出结果说明，已经成功删除了重做日志组 4 的一个成员 D:\TEMP\REDO04A.LOG。但是操作系统中和该成员对应的文件还没有被删除，需要手动删除。

在删除日志成员时，并不是所有的重做日志成员都可以删除，Oracle 还有一些限制条件，执行删除操作的一些限制如下：

- 如果要删除的日志成员是重做日志组中的最后一个有效的成员，则不能删除，如该日志组中只有一个日志成员。
- 如果该日志组正在使用，在日志切换前不能删除组中的成员。
- 如果数据库正运行在 ARCHIVELOG 模式下，并且要删除的日志成员所属的日志组没有被归档，则该组中的日志成员不能被删除。

16.4.3 重设重做日志的大小

Oracle 没有提供直接修改联机重做日志大小的方法，而是需要先删除该日志文件所在的日志组，然后再重建日志文件，通过这种方式间接地设置联机重做日志的大小。

下面给出多个实例说明重设联机重做日志大小的方法。在该实例中，需要重新设置重做日志组中日志成员的大小，将其修改为 50M。

1. 删除重做日志组

在这里假设用户的数据库上已经创建了一个重做日志组 4（如何创建重建日志组可参见第 16.3 节），该日志组中有两个日志成员，分别为 e:\redo1.log 和 e:\redo2.log。当前的日志文件信息如实例 16-16 所示。

【实例 16-16】查询当前的重做日志组信息。

```
SQL> select group#,sequence#,bytes,members,status
  2  from v$log;

    GROUP#  SEQUENCE#      BYTES    MEMBERS  STATUS
---------- ---------- ---------- ---------- -----------
         1     109063    1048576          1  ACTIVE
         2     109064    1048576          1  CURRENT
         3     109062    1048576          1  INACTIVE
         4          0   15728640          2  UNUSED
```

由该输出可以看出重做日志组 4 有两个成员，且大小都为 15M。再使用实例 16-17 查看日志文件信息。

【实例 16-17】查看日志组成员信息。

```
SQL> select *
  2* from v$logfile

    GROUP#  STATUS   MEMBER
---------- -------- ------------------------------------
         1          F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
         2          F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
         3          F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
         4          E:\REDO1.LOG
         4          E:\REDO2.LOG
```

从该输出中可以知道日志组 4 的两个日志成员分别为 E:\REDO1.LOG 和 E:\REDO2.LOG，现在要修改重做日志组 4 中日志成员的大小，修改为 50M，所以先删除该重做日志组，如实例 16-18 所示。

【实例 16-18】删除该重做日志组。

```
SQL> alter database drop logfile group 4;
alter database drop logfile group 4
```

```
*
ERROR 位于第 1 行:
ORA-01623: 日志 4 是线程 1 的当前日志 - 无法删除
ORA-00312: 联机日志 4 线程 1: 'E:\REDO2.LOG'
ORA-00312: 联机日志 4 线程 1: 'E:\REDO1.LOG'
```

上述错误说明：日志组 4 为当前重做日志组，数据库服务器正在使用该重做日志组，显然这样的日志组是不能删除的。为了验证我们的判断，通过实例 16-19 查看当前的重做日志组。

【实例 16-19】查看当前的重做日志组。

```
SQL> select group#,bytes,members,status
  2  from v$log;

    GROUP#      BYTES    MEMBERS STATUS
---------- ---------- ---------- ----------
         1    1048576          1 INACTIVE
         2    1048576          1 ACTIVE
         3    1048576          1 INACTIVE
         4   15728640          2 CURRENT
```

上述输出验证了我们的判断，日志组 4 为当前（CURRENT）数据库服务器正在使用的重做日志组，在删除该日志组前需要使用日志切换口令，切换到其他可用的重做日志组，如实例 16-20 所示。

【实例 16-20】日志切换并删除当前没使用的日志组。

```
SQL> alter system switch logfile;

系统已更改。
SQL> select group#,bytes,members,status
  2  from v$log;

    GROUP#      BYTES    MEMBERS STATUS
---------- ---------- ---------- ---------
         1    1048576          1 CURRENT
         2    1048576          1 INACTIVE
         3    1048576          1 ACTIVE
         4   15728640          2 INACTIVE
```

此时，重做日志组 4 的状态为 INACTIVE，如果在使用日志切换指令后，重做日志组 4 处于 ACTIVE 状态，则可以继续使用日志切换指令加速重做日志组 4 的归档工作，或者此时强制加入一个检查点，其指令格式为 ALTER DATABASE CHECKPOINT，使得要删除的重做日志组处于 INACTIVE 状态，因为强制检查点，使得 DBWR 保存在联机重做日志中，已经变化的内容写到数据文件中。现在可以删除该重做日志组了，如实例 16-21 所示。

【实例 16-21】删除重做日志组。

```
SQL> alter database drop logfile group 4;

数据库已更改。
```

再通过实例 16-22 验证删除结果。

【实例 16-22】验证实例 16-21 的删除结果。

```
SQL> select *
  2  from v$logfile;

    GROUP# STATUS  MEMBER
---------- ------- ----------------------------------
         1         F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
         2         F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
         3         F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
```

输出结果说明该重做日志组 4 被成功删除，下面重建该重做日志组，并注意修改日志成员的大小，如果日志成员的存储目录、文件名和先前删除的重做日志组 4 中日志成员相同，则必须先用操作系统命令删除重做日志 4 中的垃圾文件，否则在重新建立重做日志组 4 时，无法创建重做日志成员。

2. 重建重做日志组 4

修改重做日志文件的大小为 50M，如实例 16-23 所示。

【实例 16-23】重建重做日志组 4 并修改重做日志文件的大小。

```
SQL> alter database add logfile group 4
  2  ('e:\redo1.log',
  3  'e:\redo2.log')
  4* size 50M

数据库已更改。
```

【实例 16-24】验证是否添加成功，以及日志成员的大小。

```
SQL> select group#,bytes,members,status
  2  from v$log;

    GROUP#      BYTES    MEMBERS STATUS
---------- ---------- ---------- ---------
         1    1048576          1 ACTIVE
         2    1048576          1 CURRENT
         3    1048576          1 INACTIVE
         4   52428800          2 UNUSED
```

从上述输出可以看出，日志组 4 为新建的重做日志，因为其状态信息为 UNUSED，该日志组有两个日志成员，每个日志成员的大小都为 50M。此时，成功修改了重做日志组中日志成员的大小。

16.5　清除联机重做日志

在数据库服务器处于归档（ARCHIVELOG）模式时，如果当前正在使用的重做日志组中的重做日志文件损坏，则该重做日志不能完成归档工作，使得数据库因无法归档而挂起，在这种情

况下，需要通过清除联机重做日志来重新初始化联机重做日志文件，代码如下所示：

```
ALTER DATABASE CLEAR LOGFILE GROUP n
```

在执行上面的指令后，会清除日志文件。



在使用清除联机重做日志文件指令后，已经删除的重做日志组中重做日志的序列号变为 0，所以此时需要做数据库的全备份，因为 Oracle 在进行数据库恢复时，需要连续的序列号。

16.6　日志切换和检查点事件

当一组重做日志组写满时，或用户发出 alter database switch logfile 时，就会触发日志切换，此时 Oracle 寻找下一个可用的重做日志组，如果数据库处于归档模式，则在将当前写满的日志组归档完成前，不会使用新的重做日志组。

检查点事件是 Oracle 为了减少数据库实例恢复时间而设置的一个事件，当该时间发生时，LGWR 进程将重做日志缓冲区中的数据写入重做日志文件中，而同时通知 DBWR 进程将数据库高速缓存中的已经提交的数据写入数据文件，所以检查点事件越频繁，则用于数据库恢复的重做数据就越少。此时，检验点事件也会修改数据文件头信息和控制文件信息，以记录检查点的 SCN。

可以使用如下指令强制启动检查点事件。

```
alter database checkpoint
```

下面给出一个实例，先更改强制日志切换，为了加速日志切换时间，使得当前的重做日志文件处于 INACTIVE 状态，最后强制产生检查点事件。

【实例 16-25】强制日志切换并强制产生检查点事件。

```
SQL>conn /as sysdba
SQL> alter system switch logfile;

系统已更改。
SQL> alter system checkpoint;

系统已更改。
```



检查点事件不是检查点进程触发的，如果不是强制产生检查点事件，则检查点事件由 DBWR 数据库写进程触发。

16.7　使用 OMF 管理重做日志文件

Oracle 提供了自动管理重做日志文件的方法，即 OMF（Oracle Manager File），使用 OMF

时需要先设置联机重做日志文件的存储目录，而文件名由 Oracle 自动生成并维护。一旦设置了重做日志的存储目录，再使用 ALTER DATABASE ADD LOGFILE 指令就可以自动添加重做日志文件，文件大小为 100M。

【实例 16-26】设置重做日志的存储目录。

```
SQL> alter system set db_create_online_log_dest_1 = 'd:\redo1';

系统已更改。

SQL> alter system set db_create_online_log_dest_2 = 'd:\redo2';

系统已更改。

SQL>  alter system set db_create_online_log_dest_3 = 'd:\redo3';

系统已更改。
```

此时，成功设置了重做日志的存储目录。当使用 ALTER DATABASE ADD LOGFILE 指令时，Oracle 会自动在上述三个目录下创建重做日志成员，成员大小为 100M，如实例 16-27 所示。

【实例 16-27】使用 ALTER DATABASE ADD LOGFILE 指令创建重做日志成员。

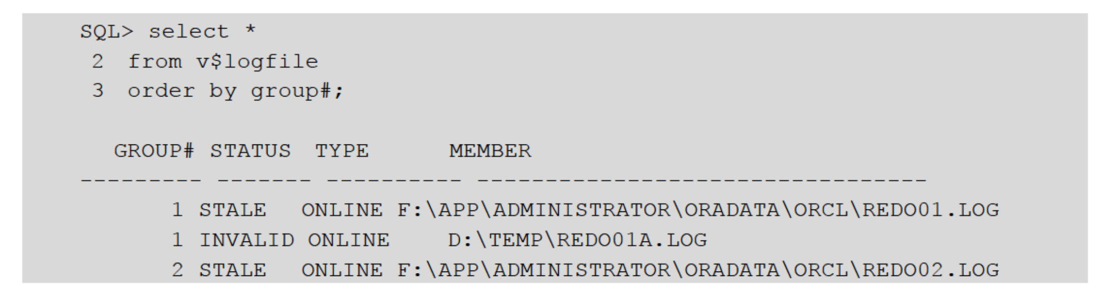
```
SQL> alter database add logfile;

数据库已更改。
SQL> select group#,sequence#,members,bytes,status,archived
  2  from v$log;

    GROUP#  SEQUENCE#    MEMBERS      BYTES  STATUS           ARC
---------- ---------- ---------- ---------- ---------------- ---
         1         39          2   10485760 INACTIVE         NO
         2         38          2   10485760 INACTIVE         YES
         3         37          2   10485760 INACTIVE         YES
         4         40          1   52428800 CURRENT          NO
         5          0          3  104857600 UNUSED           YES
```

通过上述实例说明，添加了一个重做日志组 5，Oracle 自动获得该日志组号，而且该日志组有 3 个日志成员，使用实例 16-28 来验证日志文件名。

【实例 16-28】验证日志文件名。



```
SQL> select *
 2  from v$logfile
 3  order by group#;

   GROUP# STATUS   TYPE       MEMBER
--------- -------- ---------- ----------------------------------
        1 STALE    ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO01.LOG
        1 INVALID ONLINE      D:\TEMP\REDO01A.LOG
        2 STALE    ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO02.LOG
```

```
        2 INVALID ONLINE     D:\TEMP\REDO02A.LOG
        3 STALE   ONLINE F:\APP\ADMINISTRATOR\ORADATA\ORCL\REDO03.LOG
        3 INVALID ONLINE     D:\TEMP\REDO03A.LOG
        4 CURRENT ONLINE     E:\REDO1.LOG
        4 CURRENT ONLINE     E:\REDO2.LOG
        5         ONLINE    D:\REDO1\ORA 5 54D88M00.LOG
        5         ONLINE    D:\REDO2\ORA_5_54D89500.LOG
        5         ONLINE    D:\REDO3\ORA_5_54D89000.LOG
```

显然日志组 5 有三个日志文件，文件名为 D:\REDO1\ORA_5_54D88M00.LOG、D:\REDO2\ORA_5_54D89500.LOG 和 D:\REDO3\ORA_5_54D89O00.LOG。



当删除该重做日志组 5 时，由于它是 OMF 管理的，所以该日志组下的操作系统上的重做日志文件会自动清除。

16.8　归档重做日志

归档重做日志就是联机重做日志的脱机备份，在数据库服务器处于归档模式时，发生日志切换时，数据库的归档进程 ARCH 把重做日志文件中的数据移动到归档重做日志中。归档进程在数据库服务器运行期间并不总是存在，而是当满足一定条件，如一组重做日志文件写满时启动归档进程。一旦归档完毕，归档进程自动关闭。

归档日志文件存储在参数文件 SPFILE 或 init.ora 中参数指定的位置，在 inti.ora 文件中该参数为 log_archive_dest_n。Oracle 只能把重做日志中的数据移动到磁盘上，而不能移动到磁带等存储介质上。

16.9　本章小结

本章主要讲解了 Oracle 引入重做日志的目的，以及 Oracle 复杂的重做日志文件结构，这里主要涉及两个概念，即重做日志组和重做日志成员。在理解了上述内容后，读者需要掌握如何管理重做日志组和维护重做日志文件，这是 DBA 经常使用的维护操作。日志切换和检查点事件是与重做日志文件管理相关的很重要的两个概念。需要读者认真体会，归档日志文件是重做日志文件的脱机备份，在生产数据库中要求数据库服务器处于归档模式，而设置归档模式在 Oracle 9i 和 Oracle 10g、Oracle 11g 中的方法略有不同。

第 17 章

还原数据管理

还原数据是为了实现数据更改的同时，其他用户或进程可以并发访问正在更新而没有提交的数据。除此之外，还原数据在事务恢复和事务回滚方面也发挥了作用。本章我们首先介绍 Oracle 引入还原数据的原因，然后详细介绍存储还原数据的还原段的类型，而自动还原段管理是 Oracle 推荐的管理方式，还原段逻辑地保存在还原表空间中，所以还原表空间的创建与维护也是本章介绍的重点。

17.1 引入还原数据的原因

在实际的生产数据库中，用户读取数据库中的一行数据并对其进行修改，但是此时其他用户也需要读取该行数据以实现查询需要。Oracle 数据库为了满足这种需求而设计了一种解决方案，即把用户需要修改的数据放在一个还原段中，此时除了正在修改数据的用户外其他用户只能读取该行数据在还原段中的数据。这样就实现了数据修改与数据读取的并行性，不影响多用户对数据的访问，使用图 17-1 说明这个过程。

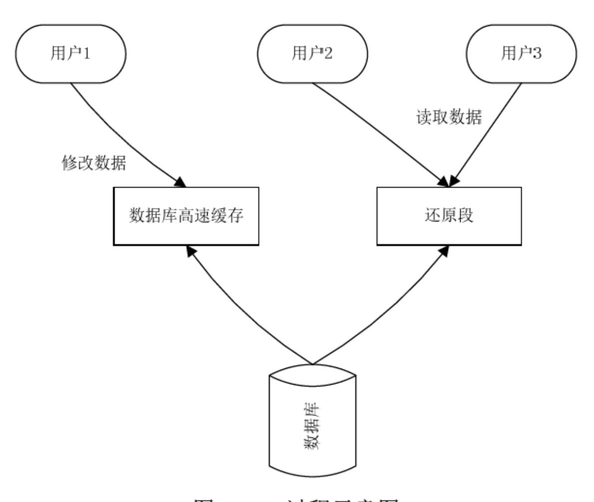


图 17-1　过程示意图

在图 17-1 中，当需要修改一行数据时，用户进程从数据库读取该行数据到数据库高速缓存中，同时该行数据的副本复制到还原段中，有用户需要访问该行数据时，可以读取还原段中的数据。

Oracle 还原段的引入确实解决了修改数据时并行读数据的问题，当用户修改数据时，该数据首先复制到还原段中，一个事务将它需要修改的全部数据放在同一个还原段中，还原段的用途，即事务恢复、事务回滚和读一致性，如图 17-2 所示。

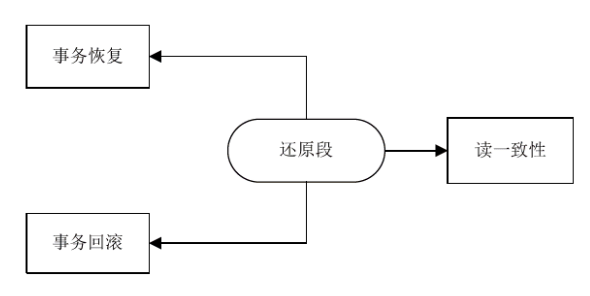


图 17-2　还原段的作用示意图

下面依次介绍还原段的几个作用。

- 事务恢复：要实现还原段进行数据恢复需要还原段上数据的变化记录在重做日志文件中。一旦某个事务执行期间数据库实例崩溃，再次启动数据库时就需要还原没有提交的数据，恢复这些数据的原始值。
- 事务回滚：如果用户更改了某行数据，而后恢复修改后且没有提交的数据，此时使用 ROLLBACK 语句回滚修改的数据，Oracle 数据库服务器就使用还原段中的数据完成数据的回滚操作。
- 读一致性：在用户修改数据期间，如果修改的数据还没有提交，则其他读取该数据的用户不应该看到这些被修改了且没有提交的数据，而应该看到这些数据的原始值，而这些数据的原始值就放在还原段中。

17.2　还原段的分类

Oracle 数据管理系统将还原段分为两大类，即：系统还原段和非系统还原段，其中系统还原段为系统表空间使用，当系统表空间中的对象发生变化时，这些对象的原始值就保存在系统还原段中，系统还原段在系统表空间中创建，即可以工作在自动模式下，也可以工作在手动模式下。

非系统还原段为非系统表空间（如用户表空间等）所使用。当一个数据库系统具有非系统表空间时，就需要至少一个非系统还原段或一个自动管理的还原表空间，其中自动管理模式由数据库服务器自动维护，但需要至少一个还原表空间，而手动管理模式需要管理员创建非系统还原段，这些手动的非系统还原段又包括两种类型，即：公有还原段和私有还原段。还原段的分类如图 17-3 所示。

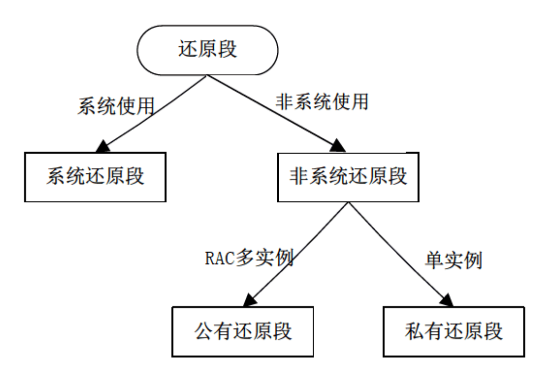


图 17-3　还原段分类示意图

在 Oracle 11g 中实现了还原段的自动管理功能（实际在 Oracle 9i 以上版本都实现了还原段的自动管理），使用自动管理的方式需要首先创建一个还原表空间，并经还原表空间告诉数据库服务器，之后的维护工作由数据库服务器自动完成。

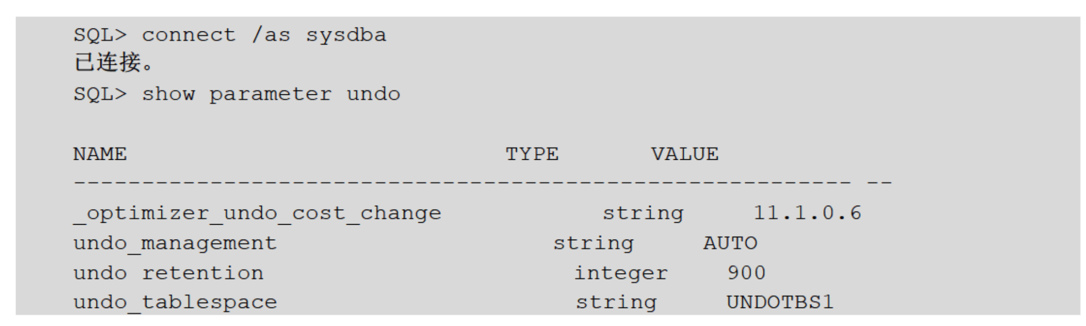
17.3　还原段的管理

在上节已经讲了在高版本的 Oracle 数据库中还原段的管理是自动维护的，所以需要为数据库服务器创建一个还原表空间作为存放还原数据的逻辑结构。

在 Oracle 11g 数据库中需要设置两个参数来设置还原段的自动管理，一个为 UNDO_MANAGEMENT，说明还原段的管理方式，它不是一个动态参数，需要在参数文件中修改，然后重新启动数据库方可生效；另一个为 UNDO_TABLESPACE，说明还原表空间的名字，该参数是动态参数，可以在数据库运行期间动态修改。

在数据库启动后，为了知道当前数据库还原段的管理方式，我们使用实例 17-1 所示的方式来查看。

【实例 17-1】查看和还原段相关的参数。



```
SQL> connect /as sysdba
已连接。
SQL> show parameter undo

NAME                                TYPE        VALUE
---------------------------------------------------------- --
_optimizer_undo_cost_change               string      11.1.0.6
undo_management                         string      AUTO
undo retention                            integer     900
undo_tablespace                           string      UNDOTBS1
```

笔者当前的数据库系统为 Oracle 11g 11.1.0.6，在实例 17-1 的输出中，我们看到当前的数据库管理方式为自动管理，而还原表空间是 UNDOOTBS1。当然也可以通过参数文件查看还原段的管理信息，如图 17-4 所示。



图 17-4 初始化参数文件中的还原段信息

其实，在 Oracle 9i、Oracle 10g 以及 11g 数据库中，还原段的默认方式都为自动管理方式，而还原表空间为 UNDOTBS1，也是在 Oracle 自动创建的一个还原表空间。当然这个还原表空间可以通过指令修改，如下所示。

```
SQL>ALTER DATABASE SET UNDO_TABLESPACE undo_tablespace_name
```

在以下几节将讲解如何创建还原表空间、如何切换到新的还原表空间以及如何删除还原表空间的问题。

17.4 还原表空间的创建

在本节将讲解如何在创建数据库后创建还原表空间，这也是在实际的生产数据库维护中经常使用的一种创建还原表空间的方式，至于在数据库创建时建立还原表空间的内容可以参考创建 Oracle 数据库一章。

下面创建一个还原表空间，使用指令 CREATE UNDO TABLESPACE，如实例 17-2 所示。

【实例 17-2】创建一个还原表空间。

```
SQL> conn system/oracle@orcl
已连接。
SQL> create undo tablespace my_undo
  2  datafile 'd:\undo\my undo.dbf'
  3  size 100M
  4  autoextend on;

表空间已创建。
```

在实例 17-2 中创建了一个还原表空间，其名字为 my_undo，该还原表空间中有一个数据文件，即 d:\undo\my_undo.dbf，大小为 100M，并且为自动扩展方式，即当还原表空间的空间不足时，该数据文件可以自动扩展。下面通过实例 17-3 查看新建的还原表空间的信息。

【实例 17-3】查看还原表空间的信息。

```
SQL> select tablespace_name,extent_management,contents,logging,status
  2  from dba_tablespaces
  3* where contents ='UNDO'

TABLESPACE NAME         EXTENT MAN CONTENTS  LOGGING   STATUS
----------------------------------------- --------------------- -
UNDOTBS1                 LOCAL        UNDO       LOGGING   ONLINE
```

```
MY_UNDO                        LOCAL          UNDO       LOGGING    ONLINE
```

从上述输出可以看出，还原表空间 MY_UNDO 已成功创建，而且处于在线状态，说明随时可以使用切换指令将还原表空间切换到 MY_UNDO。EXTENT_MANAGEMENT 为本地管理方式（默认方式），因为 LOGGING 为 LOGGING，所以该还原表空间受到重做日志的保护，可以用于数据库实例恢复。

再通过数据字典 DBA_DATA_FILES 查看关于该还原表空间的数据文件的一些信息。读者可以通过 DESC 指令查看数据字典视图 DBA_DATA_FILES 的列属性来查找自己需要的属性信息。

【实例 17-4】查看关于还原表空间的数据文件信息。

```
SQL> col tablespace_name for a15
SQL> col file_name for a20
SQL> select tablespace_name,file_name,bytes/1024/1024 MB ,autoextensible
  2  from dba_data_files
  3* where tablespace_name = 'MY_UNDO'

TABLESPACE_NAME FILE_NAME                      MB  AUT
------------------------------ ------------------------------------------
MY_UNDO         D:\UNDO\MY_UNDO.DBF          100  YES
```

实例 17-4 的查询结果说明还原表空间 MY_UNDO 中有一个数据文件，即：D:\UNDO\MY_UNDO.DBF，大小为 100M，因为 AUTOEXTENSIABLE 为 YES，所以该数据文件为自动扩展方式。



创建还原表空间的指令需要在 DBA 权限的用户模式下操作。

17.5　还原表空间的维护

在成功创建还原表空间后，可以通过指令动态地修改还原表空间的一些配置信息，如重命名还原表空间、增加数据文件、将数据文件设置为离线或在线状态等。

1. 重命名还原表空间

重命名还原表空间可使用 RENAME 指令，如将 MY_UNDO 重命名为 LIN_UNDO，如实例 17-5 所示。

【实例 17-5】重命名还原表空间。

```
SQL> alter tablespace my undo
  2  rename to lin_undo;'

表空间已更改。
```

2. 增加数据文件

使用 ADD DATAFILE 指令向还原表空间中添加数据文件，以增加还原表空间的容量，如实例 17-6 所示。

【实例 17-6】向还原表空间中添加数据文件。

```
SQL> alter tablespace my_undo
  2  add datafile 'd:\undo\lin_undo2.dbf'
  3  size 100M;

表空间已更改。
```

我们向还原表空间 MY_UNDO 中添加了一个数据文件，且大小为 100M，通过实例 17-7 验证添加结果。

【实例 17-7】验证实例 17-6 中添加数据成员的结果。

```
SQL>select tablespace_name,file_name,bytes/1024/1024 MB ,autoextensible
  2  from dba data files
  3* where tablespace_name = 'MY_UNDO'

TABLESPACE_NAME FILE_NAME                          MB AUT
---------------------------- ----------------------
MY_UNDO         D:\UNDO\MY_UNDO.DBF               100 YES
MY_UNDO         D:\UNDO\LIN_UNDO2.DBF              100 NO
```

可以看到还原表空间 MY_UNDO 有两个数据文件，其中一个数据文件 LIN_UNDO1.DBF 是刚刚添加的，其大小为 100M。注意此时该文件的 AUTOEXTENSIBLE 为 NO，所以该数据文件填满时将不能自动扩展大小，所以，需要修改该数据文件的参数设置。

3. 将数据文件修改为自动扩展模式。

我们给出实例 17-8 说明如何设置数据文件为自动扩展方式。

【实例 17-8】设置数据文件为自动扩展方式。

```
SQL> alter database
  2  datafile 'd:\undo\lin undo2.dbf'
  3  autoextend on;

数据库已更改。
```

现在已经将新添加的数据文件 lin_undo2.dbf 设置为自动扩展了，再通过实例 17-9 验证修改结果。

【实例 17-9】验证实例 17-8 的修改结果。

```
SQL>  select tablespace_name,file_name,bytes/1024/1024 MB ,autoextensible
  2   from dba_data_files
  3   where tablespace name = 'MY UNDO';
```

```
TABLESPACE_NAME FILE_NAME                              MB AUT
------------------------------ ------------------------------
MY_UNDO         D:\UNDO\MY_UNDO.DBF                    100 YES
MY_UNDO         D:\UNDO\LIN_UNDO2.DBF                   100 YES
```

此时，数据文件 D:\UNDO\LIN_UNDO2.DBF 的 AUTOEXTENSIBLE 为 YES，说明已经将该数据文件设置为自动扩展模式了。

在前面的内容中创建了一个还原表空间 MY_UNDO，并且添加了数据文件以增大该表空间的容量，而且两个数据文件都是自动扩展的方式，所以方便了数据库的维护。

17.6 还原表空间的切换

为了减少 I/O 或者避免磁盘空间受限的限制，需要切换还原表空间。在 Oracle 数据库中一个实例只允许有一个活跃的还原表空间，且可以通过动态的方式切换还原表空间，我们先查看当前还原表空间的信息，如实例 17-10 所示。

【实例 17-10】查看当前还原表空间的信息。

```
SQL> show parameter undo;

NAME                                TYPE        VALUE
----------------------------------------------------------- -------
undo_management                      string      AUTO
undo_retention                        integer     900
undo_tablespace                       string      UNDOTBS1
```

我们看到当前的还原表空间为 UNDOTBS1。下面使用实例 17-11 演示如何切换还原表空间。

【实例 17-11】切换到新建的还原表空间 MY_UNDO。

```
SQL> alter system set undo_tablespace = my_undo;

系统已更改。
```

在成功执行切换还原表空间的指令后，显示“系统已更改”，为了验证是否成功更改了当前数据库的还原表空间，可通过实例 17-12 进行验证。

【实例 17-12】验证是否成功切换了当前数据库的还原表空间。

```
SQL> show parameter undo;

NAME                                TYPE        VALUE
----------------------------------------------------------- -----------
undo_management                      string      AUTO
undo retention                        integer     900
undo_tablespace                       string      MY_UNDO
```

从实例 17-12 的输出可以看到，当前的 UNDO_TABLESPACE 为 MY_UNDO。

17.7　还原表空间的删除

当不需要还原表空间时，可以删除该表空间，但是要求该还原表空间不是当前活跃的还原表空间，如实例 17-13 所示删除表空间 MY_UNDO。

【实例 17-13】删除表空间 MY_UNDO。

```
SQL> drop tablespace my_undo;
drop tablespace my_undo
*
第 1 行出现错误:
ORA-30013: 还原表空间 'MY_UNDO' 当前正在使用中
```

显然有错误提示，说明删除的是当前正在使用的还原表空间，所以在删除前，先切换到另一个还原表空间，如实例 17-14 所示。

【实例 17-14】切换到新的还原表空间。

```
SQL> alter system set undo_tablespace = undotbs1;

系统已更改。
```

查询表空间切换是否成功，如实例 17-15 所示。

【实例 17-15】查询表空间切换是否成功。

```
SQL> show parameter undo;

NAME                                TYPE        VALUE
------------------------------------------------------------------------
undo_management                      string      AUTO
undo_retention                        integer     900
undo_tablespace                       string      UNDOTBS1
```

输出显示当前活跃的还原表空间是 UNDOTBS1，所以此时可以删除还原表空间 MY_UNDO 了，如实例 17-16 所示。

【实例 17-16】删除还原表空间 MY_UNDO。

```
SQL> drop tablespace my_undo;

表空间已删除。
```

此时虽然删除了还原表空间，但是只是删除了该表空间在数据字典中的定义，而其数据文件没有删除，所以需要手动使用操作系统指令删除该表空间中的数据文件。通过实例 17-17 验证是否删除了表空间 MY_UNDO。

【实例 17-17】验证是否删除了表空间 MY_UNDO。

```
SQL> select tablespace_name,status,contents
  2  from dba_tablespaces
```

```
  3  where contents = 'UNDO';

TABLESPACE_NAME                STATUS    CONTENTS
---------------------------------------------------------- --
UNDOTBS1                        ONLINE    UNDO
```

我们看到 CONTENTS 为 UNDO 的表空间已经没有 MY_UNDO 了，说明已经成功删除了还原表空间 MY_UNDO。切记需要手工删除该表空间中的数据文件，不然会造成垃圾文件而白白占用磁盘空间。

17.8　undo_retention 参数

在 Oracle 9i、Oracle 10g 和 Oracle 11g 中都引入了 UNDO_RETENTION 参数，该参数是一个时间值，说明当还原段中的数据在事务提交后继续保留的时间。该参数默认值为 900 秒，可以根据需要动态更改该参数值。

在还原段中的数据如果保留时间过长，而且修改数据的事务很多，则往往造成还原表空间的不足，所以需要根据业务需要和还原表空间的大小做出判断。下面演示如何更改参数 UNDO_RETENTION 的大小，如实例 17-18 所示。

【实例 17-18】更改参数 UNDO_RETENTION 的值。

```
SQL> alter system set undo retention = 1200;

系统已更改。
```

在更改了参数 UNDO_RETENTION 之后，需要验证更改结果，如实例 17-19 所示。

【实例 17-19】验证实例 17-18 的更改结果。

```
SQL> show parameter undo;

NAME                                TYPE        VALUE
------------------------------------------------------ ------
undo_management                      string      AUTO
undo_retention                        integer     1200
undo_tablespace                       string      UNDOTBS1
```

可以看到此时 UNDO_RETENTION 的 VALUE 为 1200，说明已修改成功。其实，也可以通过数据字典 v$parameter 来查询，该数据字典只有两列：NAME 和 VALUE，下面通过实例 17-20 演示如何通过数据字典 v$parameter 查询参数 UNDO_RETENTION 的值。

【实例 17-20】通过数据字典 v$parameter 查询参数 UNDO_RETENTION 的值。

```
SQL> col name for a20
SQL> col value for a30
SQL> select name,value
  2  from v$parameter
```

```
  3  where name = 'undo_retention';

NAME                 VALUE
-------------------- ----------------------
undo_retention        1200
```

该显示结果与实例 17-19 相同。

17.9　本章小结

本章讲解了还原数据管理的知识，从 Oracle 引入还原段的目的入手，介绍了还原段的三个作用：事务恢复、事务回滚和读一致性。Oracle 按照管理需要将还原段分为系统还原段和非系统还原段，并重点讲解了如何实现自动的还原段管理，围绕这一主题，我们详细介绍如何创建和维护还原表空间，其中维护还原表空间涉及重命名、为表空间增加数据文件、设置数据文件为自动扩展方式、如何切换还原表空间和删除不用的还原表空间，随后介绍了 UNDO_RETENTION 参数及其使用，最后讨论了几个重要的和还原段相关的数据字典视图。通过这些动态视图读者可以更好地了解还原段的使用情况。

第 18 章

PL/SQL语言基础

PL/SQL 语言是对 SQL 语言的功能扩充，SQL 语言适合管理关系型数据库，但是无法满足应用程序对数据更复杂的处理需求。PL/SQL 语言用于创建存储过程、函数、触发器、PL/SQL 包和用户自定义函数。Oracle PL/SQL 在企业级应用程序中的应用广泛，而且 Oracle 的一些功能部件也是使用 PL/SQL 编写的。

18.1 PL/SQL 的代码块结构

使用任何编程语言都要求一定的语言结构，通过这种结构编写具有一定功能的代码块，这种代码结构也是一种逻辑结构，Oracle 的 PL/SQL 代码块的结构如图 18-1 所示。PL/SQL 代码块由 4 部分组成，即块头区、声明区、执行区和异常区。

```
块头区
IS
    声明区
BEGIN
    执行区
EXCEPTION
    异常区
END
```

图 18-1 PL/SQL 的代码块结构

PL/SQL 语言可以编写存储过程、函数以及触发器，而这些数据库对象的创建都使用 PL/SQL 代码块结构实现。

18.1.1　块头区

块头区包含程序单元名字和参数，其中程序单元名字可以是函数（FUNCTION）、存储过程（PROCEDURE）或者包（PACKAGE），而参数具有一定的数据类型，该参数分为三类，一类是 IN 参数，该参数表示将参数传递给程序段单元，如存储过程，另一个是 OUT 参数，该参数返回给调用该程序的（如函数）对象，最后一类是双向的 IN OUT 参数。块头区的结构如下所示。

```
Program_type program_name ([parameter_name IN / OUT /IN OUT type specs,]……)
[RETURN datatype]
```

其中 Program_type 可以是 FUNCTION、PROCEDURE 或 PACKAGE，参数类型是 PL/SQL 定义的数据类型，而 specs 可以包含关键字 NOT NULL，确保该参数值一定存在，如果没有参数值则使用默认值。

对于函数 FUNCTION 而言必须返回数值，函数可以在其执行部分的任意位置使用 RETURN 关键字，返回该函数定义的返回数据类型。如下所示说明创建 PL/SQL 函数时的块头区：

```
CREATE OR REPLACE FUNCTION fun_test(f float)
RETURN float
```

上例定义了一个函数 FUNCTION，函数名为 fun_test，而传递给该函数的参数是一个浮点数 FLOAT，而该函数的返回值也是一个浮点数。

PL/SQL 过程没有返回值，创建 PL/SQL 过程（PROCEDURE）的块头区如下所示：

```
CREATE OR REPLACE PROCEDURE pro_test(name IN varchar2)
```

上述创建了 PL/SQL 过程 PROCEDURE，过程名为 pro_test，参数名为 name，参数类型为 varchar2，而该参数输入给该过程使用。

上面创建了 PL/SQL 程序单元的块头区，其实也可以创建不包含块头的 PL/SQL 程序单元，如过程等，这样就可以内嵌的使用这些匿名程序单元，这样的程序单元不好维护，且其他程序单元无法调用它，在实际使用中最好不要使用匿名块的方式。

18.1.2　声明区

声明区的目的是声明一些变量，这些变量在块内是有效的，变量的数据类型是任何 Oracle 定义的数据类型，如 VARCHAR2、CHAR、NCHAR、NUMBER 等。在变量声明中可以使用 CONSTRAINT 约束，对变量的值做出限制，如不允许为 NULL 值等。

与在其他编程语言（如 C++或 Java 中一样），PL/SQL 语言允许使用 CONSTANT 常量来标识一个变量，该变量一旦赋值，则在整个程序单元执行期间均保持不变。在变量声明时，可以给变量赋值，赋值运算符为“:=”，也可以使用 DEFAULT 关键字为变量或常量赋值。

声明一个字符变量：

```
Var_name VARCHAR2(20);
```

声明一个带约束的字符变量：

```
Var_name VARCHAR2(20) NOT NULL;
```

声明一个常量且赋初始值：

```
Var_name CONSTANT VARCHAR2(20) := 'China';
```

声明一个变量且使用 DEFAULT 关键为变量赋值：

```
Var_name INTEGER DEFAULT 3.1415925;
```

PL/SQL 的声明区在块头区的后面，使用 IS 说明后面是声明区，也可以使用 DECLARE 关键字在程序单元的任意位置声明变量，如下所示：

```
DECLARE
   Var_name [CONSTRAINT] datatype [(constraint)] [:= value];
```

18.1.3 执行区

PL/SQL 的执行块完成该程序单元的功能逻辑，即该程序单元的行为定义部分，在执行部分可以使用流程控制以及复杂的算法，是 PL/SQL 程序单元的主体部分。

执行区使用 BEGIN 和 END 标识，其中 BEGIN 标识执行区的开始，而 END 标识执行区的结束。Oracle 对执行区的要求是至少包含一条执行语句，甚至可以是 NULL，但是不能为空，并且执行区在 PL/SQL 程序单元中是必须定义的，执行区的结构如下所示。

```
BEGING
 //LOGIC STATEMENTS
END;
```

18.1.4 异常区

同任何计算机编程语言一样，有效地处理异常是该语言健壮性的体现，在 PL/SQL 语言中设计了异常区，该区块位于 PL/SQL 块结构 END 关键字之前，用于捕获关键字 END 之前的 PL/SQL 块抛出的异常并获得处理。

在声明区一般需要定义异常变量，在 PL/SQL 块内出现异常的地方抛出异常，且在异常区捕获该异常并处理，异常区结构如下所示。

```
EXCEPTION
         WHEN exception_name1
         THEN
                  Handl error1;
WHEN exception_name2
         THEN
                  Handl error2;
         WHEN others
              Default error handling;
```

EXCEPTION 说明下面是一个异常区，而 WHEN 后为一个具体的异常，THEN 后是对该异常的处理代码，一个异常区可有多个 WHEN、THEN 的搭配使用来处理不同的异常。

为了说明异常区的使用现给出一个实例，该实例中声明了自己的异常，如实例 18-1 所示。

【实例 18-1】创建异常区示例程序。

```
CREATE OR REPLACE PROCEDURE pro test(f float)
IS
        var_name varchar2(20);
//定义异常对象
        exception1 EXCETION;
BEGIN
        Statement1;
BEGING
    Statement2;
//抛出在执行 Statement2 语句时的异常
Raise exception1;
        EXCEPTION
        WHEN exception1
Handling errors;
END;
Statement3;
//在 statement1 和 statement3 中抛出的异常在下面这个异常区中处理。
EXCEPTION
        WHEN OTHERS
            handling errors;
END;
```

实例 18-1 中异常的执行过程：该 PL/SQL 过程执行 statement1，如果此时发生异常，则在最后的异常处理区块中处理，接着执行区块内的一个 BEGIN…END 区块，如果 Statement2 发生异常，则抛出异常 exception1，而该异常在随后的异常处理区块中处理，接着执行 Statement3，如果此时发生异常，则在该语句随后的 EXCEPTION 区块中处理。

18.2　PL/SQL 的流程控制语句

PL/SQL 作为一种数据库编程语言，必须通过程序的流程控制实现模块的功能逻辑，为了实现流程控制，Oracle PL/SQL 提供了条件语句、循环语句、分支语句等，本节将详细介绍这些流程控制语句。

18.2.1　IF 条件语句

条件语句用于逻辑判断，在日常生活中经常会遇到这样的情况：如果这个周末天气好就去爬香山，否则就呆在家里看电影。在 PL/SQL 语言中使用条件语句来实现程序自身地判断。

条件语句的语法格式如下所示：

```
IF condition1
    THEN
        Logical statement;
```

```
END IF;
```

这种条件语句首先判断 condition1 是否成立，如果成立则执行 THEN 后的代码逻辑，然后结束这个逻辑判断。

条件语句也可以是如下形式：

```
IF condition1
    THEN
        Logical statement1;
    ELSE
        Logical statement2;
END IF;
```

这种条件语句首先判断 condition1 是否成立，如果成立则执行 THEN 后的代码逻辑，否则，执行 ELSE 后的代码逻辑，然后结束这个逻辑判断。

在上面介绍的两种条件语句中对初始条件只有一次 IF 判断，其实也可以有多次判断，此时使用 ELSE IF，如下所示。

```
IF condition1
    THEN
        Logical statement1;
ELSE IF condition2
        Logical statement2;
ELSE
        Logical statement3;
END IF;
```

这种条件语句首先判断 condition1 是否成立，如果成立则执行 THEN 后的代码逻辑，接着判断条件 2，即 condition2 是否成立，如果成立则执行其随后的代码逻辑，否则，执行 ELSE 后的代码逻辑，然后结束这个逻辑判断。


说 明

在 PL/SQL 的条件语句中，条件为一个布尔表达式，该表达式要么为真 TRUE，要么为假 FALSE，注意必须使用 END IF 结束该逻辑判断。

18.2.2 CASE 条件语句

CASE 语句是一种条件语句，该语句中的条件可以是布尔值，也可以是其他值，如字符串或者数字等。CASE 条件语句的结构如下所示：

```
CASE expression
WHEN condition1
          THEN logical statement1
    WHEN condition2
           THEN
              BEGIN
 logical statement2_1;
```

```
logical statement2_2;
              END;
    WHEN condition3
           THEN logical statement3
......
ELSE
            DEFAULT logical statement;
END CASE;
```

在 CASE 条件判断中，首先计算 CASE 后的 expression 的计算结果，然后将该结果依次和随后的 WHEN 语句后的条件进行匹配，如果找到某个匹配的值，则执行该 WHEN 子句中 THEN 后的代码逻辑，如果最后没有找到则执行 ELSE 后的默认代码逻辑。



在 CASE 条件语句中务必给出 ELSE 子句，并且自己的代码逻辑是有效地，否则如果 WHEN 语句没有匹配选项，Oracle 会抛出异常。

18.2.3　循环语句

循环语句用于重复地执行一个代码逻辑，如重复访问一个数组，在数据库中依次检索表中的每条记录等，使用循环可以方便地实现这些操作。Oracle 提供了三种循环结构，即：LOOP、FOR…LOOP 和 WHILE…LOOP。下面依次介绍这三种循环结构。

1. LOOP 循环

这是最基本的循环结构，其实就是一层循环，当 LOOP 后的代码执行完毕后，遇到 EXIT 就退出循环，其结构如实例 18-2 所示。

【实例 18-2】LOOP 循环结构。

```
LOOP
    Logical statement
    EXIT [ WHEN condition];
END LOOP;
```

LOOP…EXIT 循环结构的使用如实例 18-3 所示。

【实例 18-3】使用 LOOP 循环结构的示例。

```
create or replace procedure protest1
is
begin
   loop
    dbms_output.put_line('hello1:-)');
    dbms_output.put_line('hello2:-)');
    dbms_output.put_line('hello3:-)');
    exit;
   end loop;
END;
/
```

将该上例在 Windows 的记事本中编辑，并保存为 protest.sql 文件，然后如实例 18-4 所示执行该脚本文件。

【实例 18-4】执行 protest.sql 脚本文件。

```
SQL> @f:\protest.sql;

过程已创建。
```

执行存储过程 protest1，观察其输出，如实例 18-5 所示。

【实例 18-5】执行存储过程 protest1 观察 LOOP 循环结构。

```

SQL> execute protest1
hello1:-)
hello2:-)
hello3:-)

PL/SQL 过程已成功完成。
```

从输出可以知道 LOOP 循环只执行了一次，即依次输出三个 DBMS_OUTPUT 包的 PUT_LINE 函数，然后退出循环。

2. FOR…LOOP 循环

这种结构的循环首先使用 FOR 语句确定此循环时，然后执行循环体，其语法结构如下所示：

```
FOR counter [REVERSE] low……high
LOOP
   Logic
END LOOP;
```

使用这种循环使得计数器 counter 从 low 开始计数一直到 high 值，然后结束循环。



high 和 low 的值可以是确定的值，也可以是变量。

3. WHILE…LOOP 循环

这种循环结构首先判断 WHILE 后的条件，如果条件满足则执行 LOOP 循环，如果不满足，则不进行 LOOP 循环，其逻辑结构如下所示。

```
WHILE condition
LOOP
  Logic
END LOOP;
```

其中，condition 是否执行 LOOP 循环的条件，而 Logic 是 LOOP 循环的代码主体部分。LOOP 表示循环的开始，而 END LOOP 表示退出循环。

18.2.4　分支语句

分支语句通过 GOTO 语句实现，其作用是在代码执行过程中，出于某种需要从当前的执行代码跳转到其他代码块，这样使得程序的逻辑流程指向程序的其他部分。其语法结构如下所示：

```
......
GOTO  LABEL
  Logic1
......
<<LABEL>>
Logic2
......
```



实际上在使用分支语句时使得程序流指向程序中的其他部分，而且不会返回到原来跳转的位置，这样很容易破坏程序的整体功能，在程序设计中尽量不要使用这种跳转语句，可以使用条件判断语句代替。

18.3　PL/SQL 的创建过程

在第 18.1 节介绍了使用 PL/SQL 语言编写程序单元的代码块结构，它适用于存储过程、触发器、函数、游标这样的程序单元。而要实现复杂的程序逻辑，必须使用流程控制语句，本节给出一个具体的创建存储过程的实例，通过这个实例读者可以清楚地理解 PL/SQL 的代码块结构以及如何使用一些条件逻辑语句，编写完整可运行的 PL/SQL 过程。

要创建 PL/SQL 过程可以采用两种方式，一种是采用 Windows 提供的记事本工具编写过程的脚本文件，另一种是使用 iSQL*Plus 或 SQL*Plus 直接在用户模式下编程。其步骤是创建存储过程→编译存储过程→执行存储过程。

18.3.1　开始编程

创建过程开始于对过程的初始定义，如实例 18-6 所示。

【实例 18-6】定义存储过程。

```
CREATE OR REPLACE PROCEDURE selectemp(employeeno IN INTEGER)
```

上述语句的目的是创建一个过程 PROCEDURE，过程名为 selectemp，而该过程参数名为 employeeno，数据类型为 INTEGER，该参数输入给该过程，注意 IN 关键字的作用。下面几节将依次向该过程加入其他区块，并在各自的区块内定义变量或者程序逻辑等。

18.3.2　创建变量

创建变量是在 PL/SQL 代码块结构的声明区完成的，注意此时使用了 IS 关键字，其后是变

量声明区。如实例 18-7 所示。

【实例 18-7】向过程添加变量声明。

```
CREATE OR REPLACE PROCEDURE selectemp(employeeno IN INTEGER)
IS
  employeename varchar2(20);
  employeejob       VARCHAR2(9);
  employeehiredate  DATE;
  employeesal        NUMBER(7,2);
```

此时，向该过程添加了变量声明，该区块定义了 4 个变量，即 employeename（数据类型为 VARCHAR2(20)）、employeejob（数据类型为 VARCHAR2(9)）、employeehiredate（数据类型为 DATE）、employeesal（数据类型为 NUMBER(7,2)）。分别存储从表 EMP 中获得的四个员工属性值。

18.3.3 添加执行代码

本节将介绍如何添加执行代码，即实现 PL/SQL 代码块结构的执行区逻辑，如实例 18-8 所示，读取该员工的 4 个属性，即员工名（ENAME）、工作岗位（JOB）、雇用日期（HIREDATE）和薪水（SAL），并将这 4 个属性值赋予声明区的 4 个变量。然后从标准输出装置输入读取到的信息。

【实例 18-8】向过程添加执行代码。

```
CREATE OR REPLACE PROCEDURE selectemp(employeeno IN INTEGER)
IS
  employeename varchar2(20);
  employeejob       VARCHAR2(9);
  employeehiredate  DATE;
  employeesal        NUMBER(7,2);

  BEGIN

     SELECT ename ,job,hiredate,sal
     INTO   employeename,employeejob,employeehiredate,employeesal
     FROM scott.emp
     WHERE empno = employeeno;
     DBMS_OUTPUT.put_line('员工姓名'
                         ||employeename
                         ||'工作岗位'
                         ||employeejob
                         ||'雇佣日期'
                         ||employeehiredate
                         ||'薪水'
                         ||employeesal);
  END;
```

此时，已成功完成了过程 selectemp 的执行，此时可以编译并执行该过程，但是如果出现异

常该怎么办呢？此时还无法处理，所以继续向过程添加异常处理区块。

18.3.4　处理异常

异常区位于 END 关键之前，并且其处理范围是此 BEGIN…END 之间的异常。在过程创建中，没有声明异常变量，此时使用 WHEN OTHERS…THEN 来处理异常，如实例 18-9 所示。

【实例 18-9】向过程添加异常处理代码。

```
CREATE OR REPLACE PROCEDURE selectemp(employeeno IN INTEGER)
IS
  employeename varchar2(20);
  employeejob      VARCHAR2(9);
  employeehiredate  DATE;
  employeesal       NUMBER(7,2);

  BEGIN

      SELECT ename ,job,hiredate,sal
      INTO   employeename,employeejob,employeehiredate,employeesal
      FROM scott.emp
      WHERE empno = employeeno;
      DBMS_OUTPUT.put_line('员工姓名'
                           ||employeename
                           ||'工作岗位'
                           ||employeejob
                           ||'雇佣日期'
                           ||employeehiredate
                           ||'薪水'
                           ||employeesal);
EXCEPTION
     WHEN OTHERS
      THEN
      DBMS_OUTPUT.put_line('ERRORS!!! ');

  END;
  /
```

此时，一旦在 BEGIN…END 之间发生异常，则在上述的 EXCEPTION 区块处理，此时，从标准装置输入一个错误提示 ERRORS。

在过程定义的最后输入了一个“/”符号，表示编译该过程，如果在脚本文件中不使用该符号，则在执行脚本文件时，需要显式的输入该符号来执行过程的编译。

18.4　PL/SQL 的编译与执行

在实例 18-9 中，创建了完整的存储过程，该过程的名字为 selectemp，将该文件在记事本中

编辑完成，如图 18-2 所示。

然后继续将该文件保存在 F 盘的根目录下，保存的文件名为 selectemp.sql（当然，该名字可以和存储过程名不同），后缀.sql 说明该文件是 Oracle 的脚本文件，如图 18-3 所示。

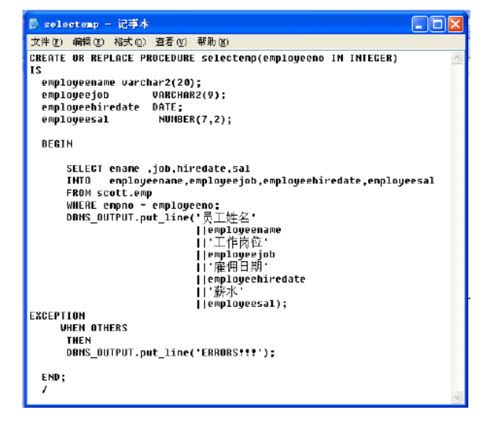


图 18-2　在记事本中编辑过程定义脚本

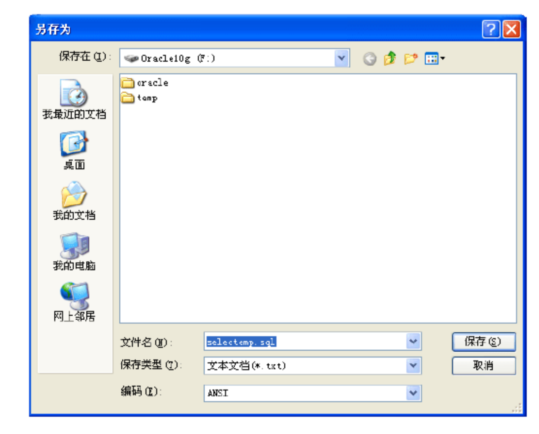


图 18-3　保存过程为脚本文件

在成功保存后，读者可以使用操作系统工具查询该文件的信息，如使用资源管理器，如图 18-4 所示。

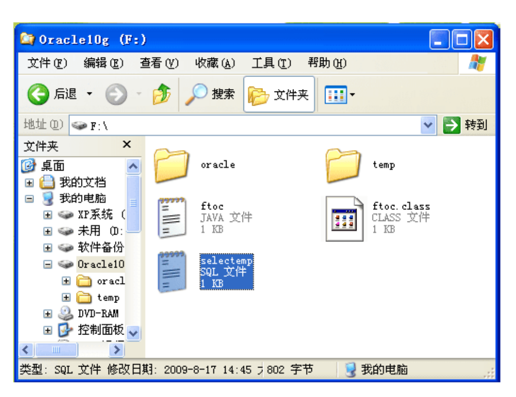


图 18-4　通过资源管理器查看保存的过程脚本文件

然后登录数据库到 SCOTT 用户模式执行该脚本文件，如实例 18-10 所示。

【实例 18-10】编译并创建过程脚本文件。

```
SQL> @f:\selectemp.sql;

过程已创建。
```

此时，提示成功创建该过程，可以使用数据字典 USER_PROCEDURES 来查看该过程的信息，如实例 18-11 所示。

【实例 18-11】通过数据字典 USER_PROCEDURES 查看过程 selectemp 的信息。

```
SQL> conn scott/oracle@orcl
```

```
已连接。
SQL> select object_name
  2  from user_procedures;

OBJECT_NAME
------------------------------
FTOC
SELECTEMP
PROTEST1
ADDSTUDENT
PROTEST
FTOCORACLE

已选择 6 行。
```

显然一旦过程编译并创建成功，在数据字典 USER_PROCEDURES 中，记录该对象信息。

【实例 18-12】执行存储过程。

```
SQL> execute selectemp(7654)
员工姓名 MARTIN 工作岗位 SALESMAN 雇佣日期 28-9 月 -81 薪水 1250

PL/SQL 过程已成功完成。
```

在关键字 EXECUTE 后输入存储过程名字，并输入参数，过程执行结果输出了查询到的员工信息。也可以使用 BEGIN…END 块调用存储过程，如实例 18-13 所示。

【实例 18-13】使用 BEGIN…END 执行存储过程。

```
SQL> begin
  2  selectemp(7369);
  3  end;
  4  /
员工姓名 SMITH 工作岗位 CLERK 雇佣日期 17-12 月-80 薪水 800

PL/SQL 过程已成功完成。
```

如果执行过程中出现异常，又是什么情况呢？现在给出一个实例，输入一个不存在的员工号，查看输出结果。

【实例 18-14】存在异常的执行过程。

```
SQL> execute selectemp(10000);

PL/SQL 过程已成功完成。
```

发现此时过程成功执行，但是没有输出任何信息，读者应该感到奇怪了，既然过程顺利执行，而且 EMP 表中不存在员工号为 10000 的员工记录，应该发生异常，而异常在哪里处理呢？其实，这是 Oracle 对我们的迷惑，读者可启动一个参数：

```
SQL> set serveroutput on
```

然后，执行该过程，如下所示：

```
SQL> execute selectemp(10000);
ERRORS!!!
PL/SQL 过程已成功完成。
```

可见，过程捕获并处理了异常，因为此时输出了 ERRORS！这是异常处理过程定义的，一旦发生异常，则输出一个错误提示。

18.5 存储过程的授权

存储过程作为一个数据库对象，其他用户必须拥有执行该过程的权限才可以使用，如先创建一个用户 cat，该用户的密码为 miao，如实例 18-15 所示。

【实例 18-15】创建用户 cat。

```
SQL> conn system/oracle@orcl
已连接。
SQL> create user cat
  2  identified by miao;

用户已创建。
SQL> grant create session to cat;

授权成功。
```

上例创建了用户 cat，并赋予该用户创建会话的权利，下面尝试使用 SCOTT 用户模式下创建的过程 selectemp。

【实例 18-16】在新创建的用户 CAT 下尝试执行过程 selectemp。

```
SQL> conn cat/miaomiao@orcl
已连接。
SQL> execute scott.selectemp(7369);
BEGIN selectemp(7369); END;

      *
第 1 行出现错误:
ORA-06550: 第 1 行, 第 7 列:
PLS-00201: 必须声明标识符 'SELECTEMP'
ORA-06550: 第 1 行, 第 7 列:
PL/SQL: Statement ignored
```

此时，用户 CAT 无法执行过程 selectemp，因为 SCOTT 用户模式下创建的过程没有将执行权利赋予 CAT 用户，而 CAT 用户也不具有 SCOTT 用户的权限。下面将执行过程 selectemp 的权限赋予所有用户，即公开它的执行权。

【实例 18-17】将过程 selectemp 的执行权限授权给所有用户。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> grant execute on selectemp to public;
```

然后在 CAT 用户模式下，执行在 SCOTT 用户模式下创建的过程 selectemp，如实例 18-18 所示。

【实例 18-18】在用户 CAT 模式下执行授权的过程 selectemp。

```
SQL> conn cat/miao@orcl
已连接。
SQL> set serveroutput on
SQL> execute scott.selectemp(7369);
员工姓名 SMITH 工作岗位 CLERK 雇佣日期 17-12 月-80 薪水 800

PL/SQL 过程已成功完成。
```

18.6　本章小结

本章介绍了 PL/SQL 语言，它是对 SQL 语言的扩展，Oracle 使用 PL/SQL 语言可以创建不同的数据库对象，如存储过程、函数和触发器等，而这些对象的创建都具有特定的代码块结构，所以在本章开始讲解了这种块结构的组成，在读者掌握了 PL/SQL 语言的基本语法、数据类型和流程控制语句后，就可以使用这种结构创建不同的数据库对象，为了说明 PL/SQL 语言如何创建数据库对象，在本章最后还介绍了创建存储过程的详细方法。

第 19 章

存储过程、函数和游标

存储过程、函数和游标都是数据库对象，存储过程是数据服务器内的一段使用 PL/SQL 语言编写的程序单元，具有 EXECUTE 权限的用户可以显式地调用过程。函数同样保存在数据库中，它可以直接在 SQL 语句中调用，或者从存储过程中调用，函数必须有返回值。而游标用于依次访问一组记录集合，它类似于一个指针依次指向记录集合中每个记录的地址，从而遍历整个记录。本章将依次介绍这三种数据库对象。

19.1 存储过程

19.1.1 存储过程概述

存储过程是保存在数据库服务器上的程序单元，这些程序单元在完成对数据库的重复操作时很有用，存储过程使用 PL/SQL 语言编写，也可以使用 Java 语言编写，存储过程被显式地调用完成过程定义的计算任务。存储过程可以接收各种 Oracle 定义的参数，用户可以在 SQL*Plus 或者任何可以执行 SQL 语句的接口处执行 PL/SQL 过程，一旦过程被创建则在数据字典中记录该数据库对象信息，其数据库对象类型为 PROCEDURE。

一个存储过程由三部分组成，即声明区、子程序区和异常处理区。其组成结构图如图 19-1 所示。

其中，声明区位于 PROCEDURE 和 BEGIN 之间，在声明区用来定义变量，如下所示。

```
CREATE OR REPLACE PROCEDURE PROTEST
IS
--声明变量
   xxx number ;
   yyy varchar2(20) :='oracle';
--声明 REF 游标
type empcursor is ref cursor;
--异常对象
read_disk_refused exception;
--内嵌函数或其他存储过程。
FUNCTION foo RETURN BOOLEAN IS
```

```
BEGIN
RETURN (XXX>1000);
END IF;
BEGIN
……
END;
```

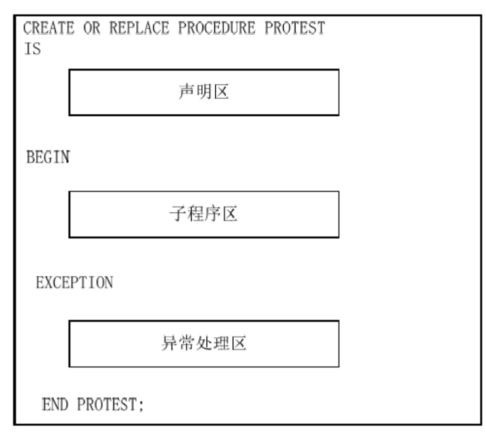


图 19-1　存储过程的组成图

在声明区中，声明了三个变量，即 NUMBER 类型、VARCHAR2 类型和 REF 游标，还声明了一个内嵌函数。

子程序区，即 BEGIN…END 之间的部分，这部分包括 PL/SQL 代码逻辑，其中的代码逻辑对于声明区是可见的，这部分是存储过程执行其功能的主体部分，并且在过程的定义中，子程序部分是不可缺少的，要求在 BEGIN…END 之间至少有一条 PL/SQL 语句，可以是 NULL，如下所示。

```
CREATE OR REPLACE PROCEDURE PROTEST
IS
BEGIN
--此处是 PL/SQL 代码逻辑
logical statement;
END;
```

异常处理区处理在子程序执行中发生的异常，如下所示。

```
CREATE OR REPLACE PROCEDURE PRO(employee age,IN NUMBER)
IS
    age    NUMBER;
    mydate DATE;
    dateexp EXCEPTION;
 BEGIN
   mydate := 'TIME_STRING';
   age := employee age;
   IF age >150
THEN
```

```
RAISE dateexp;
  EXCEPTION
   WHEN dateexp THEN
      handling dateexp;
   WHEN OTHERS   THEN
      handling other exps;
END;
```

在上例中，显式地给出了一个异常 dateexp，该异常在过程的声明区中声明，当 age>150 时就抛出 dateexp 异常，在异常处理区 EXCEPTION 后处理，注意 WHEN 子句对应了异常类型，如果是其他类型，则默认在 WHEN OTHERS THEN 后处理。

19.1.2 存储过程的创建

创建存储过程可以使用 SQL*Plus 工具，也可以在 Windows 的记事本中编辑，当使用记事本编辑存储过程时，需要将它保存为一个.SQL 脚本文件，最后使用 SQL*Plus 执行该脚本文件。使用脚本文件的方式修改起来比较方便。

当然也可以使用 SQL*Plus 工具直接输入创建过程的指令。下面使用记事本工具定义一个存储过程。该存储过程的名字为 insert_dept，作用是向 DEPT 表中插入数据，文件内容如图 19-2 所示。

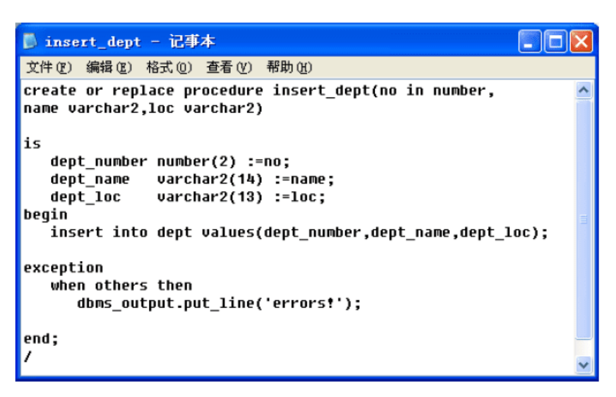


图 19-2　记事本中编辑的存储过程

将其保存为一个.SQL 脚本文件。然后执行该脚本文件创建过程，此时该存储过程作为数据库对象保存在数据字典中。当前模式下就可以使用该存储过程。保存该文件名为 insert_dept.sql，并保存在 F 盘的根目录下，然后执行该脚本文件，如实例 19-1 所示。

【实例 19-1】执行脚本文件 insert_dept.sql。

```
SQL> @f:\insert dept.sql;

过程已创建。
```

使用数据字典 USER_OBJECTS 查看存储过程 INSERT_DEPT 是否创建成功，如实例 19-2 所示。

【实例 19-2】使用数据字典查看存储过程 INSERT_DEPT 的信息。

```
SQL> col object_name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type = 'PROCEDURE'
  4* and object_name LIKE 'INSERT%'

OBJECT_NAME            OBJECT_TYPE        CREATED       STATUS
------------------------------------ -------------------------------- --
INSERT_DEPT             PROCEDURE          20-8月 -09      VALID
```

从输出可以看出，存储过程 INSERT_DEPT 已创建成功，而且 STATUS 为 VALID，所以该存储过程是有效地，当前可以使用。

19.1.3　存储过程的注意事项

在创建存储过程时，必须保证当前的用户具有创建存储过程的权限，即 CREATE PROCEDURE 和 CREATE ALL PROCEDURE。前者说明当前用户只能创建自己的存储过程，而后者表示当前用户可以创建任何用户模式下的存储过程。下面查看一下 SCOTT 用户的系统权限，如实例 19-3 所示。

【实例 19-3】查看 SCOTT 用户的系统权限。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> select *
  2  from user_sys_privs;

USERNAME              PRIVILEGE                  ADM
------------------------------------ ---------------
SCOTT                    UNLIMITED TABLESPACE     NO
SCOTT                    CREATE TRIGGER           NO
SCOTT                    CREATE ANY TRIGGER       NO
```

可见当前用户 SCOTT 没有创建存储过程的权限，所以需要 DBA 授权给 SCOTT 用户，如实例 19-4 所示。

【实例 19-4】向 SCOTT 用户授权创建存储过程。

```
SQL> conn system/oracle@orcl
已连接。
SQL> grant create procedure to scott;

授权成功。
```

输出显示授权成功，此时再次使用数据字典 USER_SYS_PRIVS 查看用户 SCOTT 的系统权限，如实例 19-5 所示。

【实例 19-5】查看 SCOTT 用户是否具备创建存储过程的权限。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> col username for a15
SQL> col privilege for a30
SQL> run
  1  select *
  2* from user_sys_privs

USERNAME        PRIVILEGE                         ADM
--------------------------- ------------
SCOTT           CREATE PROCEDURE                  NO
SCOTT           UNLIMITED TABLESPACE              NO
SCOTT           CREATE TRIGGER                    NO
SCOTT           CREATE ANY TRIGGER                NO
```

从输出的第一行可以看出用户 SCOTT 具有 CREATE PROCEDURE 的权限。当然 DBA 也可以授予 SCOTT 用户创建任何模式下的存储过程的权利，如实例 19-6 所示。

【实例 19-6】授予用户 SCOTT 创建任何模式下的存储过程的权利。

```
SQL> grant create any procedure to scott;

授权成功。
```

读者可以通过数据字典 USER_SYS_PRIVS 查看是否成功授权。

19.2　函数

19.2.1　函数概述

函数是执行某种功能的代码实体，用户调用函数输入适当的参数或者不输入参数，函数将执行计算过程，输入计算结果，函数的功能模拟如图 19-3 所示。



图 19-3　函数功能示意图

函数执行某种计算，这种计算方法由函数的功能决定。Oracle 提供了丰富的函数，用于扩展数据库的功能，如字符处理函数 LOWER()、数学运算函数 COUNT(*)、ACOS(n)计算参数的反余弦角度值等。

虽然，Oracle 定义了大量的函数，用于方便用户管理和维护数据库，但是毕竟现实的需求是多种多样的，所以 Oracle 允许用户自定义函数。本节只介绍如何使用 PL/SQL 扩展语言自定义函数。

19.2.2　函数的定义和使用

本节先通过一个实例感受如何定义和使用函数，接着介绍自定义函数的详细语法规则，然后给出一个作用于表的函数实例，最后介绍如何在 Oracle 中使用 Java 语言创建函数。

1. 自定义函数实例

下面开始自定义函数 area，该函数的作用是计算给定半径的圆的面积，用户调用该函数时输入半径参数，即可得到计算结果。创建函数的过程如实例 19-7 所示。

【实例 19-7】创建自定义函数 area。

```
SQL> CREATE OR REPLACE FUNCTION area(f float)
  2  RETURN float
  3  IS
  4  BEGIN
  5  RETURN 3.14*(f*f);
  6  END area;
  7  /

函数已创建。
```

因为函数是一种 PL/SQL 程序单元，所以其定义满足 PL/SQL 代码块结构的定义，首先声明创建函数 FUNCTION，即 CREATE OR REPLACE FUNCTION，随后是函数名以及函数参数。IS 后可以有变量声明，但不是必须的，随后是执行部分。最后的“/”表示编译该函数。

一旦函数创建成功，就可以使用该函数，为了确保已经成功创建函数 area，可使用数据字典 USER_OBJECTS，如实例 19-8 所示。

【实例 19-8】查询当前用户所拥有的函数。

```
SQL> col object name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3* where object_type = 'FUNCTION'

OBJECT_NAME          OBJECT_TYPE          CREATED        STATUS
----------------------------------- ---------------------------------
FTOCORACLE           FUNCTION            15-8 月 -09     VALID
AREA                  FUNCTION            18-8 月 -09     VALID
```

可见函数 area 创建成功，且状态为 VALID，下面就使用该函数计算给定半径的圆的面积，如实例 19-9 所示。

【实例 19-9】调用函数 area。

```
SQL> select area(4) from dual;

   AREA(4)
-------------------
     50.24
```

2. 自定义函数的语法规则

在上节中读者已经体验了如何创建一个函数，以及如何查看和调用已经创建的函数，创建自定义函数的语法如下所示。

```
CREATE [OR REPLACE] FUNCTION function_name
 ([argument [{IN|OUT}] datatype ][,…])
 RETURN datatype {IS|AS}
function_body
```

首先是指定创建或替换函数，接着是函数名和参数，参数可以是输入给该函数的（IN），也可以是输出给其他对象的（OUT），参数的个数没有限制，当然也可以没有参数，但是函数必须有返回值，需要在函数定义中明确地指定返回的数据类型 RETURN datatype。

3. 创建作用于表的函数

在学习了自定义函数的语法规则后，下面创建一个操作表对象的函数。该函数的作用是通过一个员工号，读取员工的工资和姓名，如果该员工的工资大于 4000 则输出该员工的工资数和姓名，然后返回一个变长字符变量，如实例 19-10 所示。

【实例 19-10】创建操作表对象的函数 findwealthier。

```
SQL> create or replace function findwealthier(empnumber number)
  2  return varchar2 is
  3     salary number;
  4     employeename varchar2(20);
  5   begin
  6     select ename,sal into employeename,salary
  7     from emp
  8     where empno = empnumber;
  9    if salary>=4000 then
 10       return 'make good salary';
 11       dbms_output.put_line('salary is'
 12                          ||salary
 13                          ||'name is '
 14                          ||employeename);
 15     else
 16       return 'bad';
 17       dbms output.put line('salary is'
 18                          ||salary
 19                          ||'name is '
 20                          ||employeename);
 21    end if;
 22  end findwealthier;
 23  /

函数已创建。
```

同样需要验证是否成功创建该函数，如实例 19-11 所示。

【实例 19-11】查询函数 findwealthier 是否创建成功。

```
SQL> select object_name,object_type,created
  2  from user_objects
  3  where object_type = 'FUNCTION'
  4  and object name like 'FIND%';

OBJECT_NAME          OBJECT_TYPE         CREATED
----------------------------------- ----------------------------------
FINDWEALTHIER        FUNCTION            18-8 月 -09
```

从上例输出可以看出函数 FINDWEALTHIER 已成功创建，下面可以使用该函数了，执行函数 FINDWEALTHIER 如实例 19-12 所示。

【实例 19-12】执行作用于表的函数 findwealthier。

```
SQL> select findwealthier(7839) from dual;

FINDWEALTHIER(7839)
-----------------------------------------------
make good salary

salary is5000name is KING
```

在函数参数中，输入了参数 7839，该参数表示一个员工号，而查询结果表明，该员工是高收入员工，该员工月薪为 5000，姓名为 KING。显然通过调用函数方便了用户操作，用户可以灵活地使用自定义函数根据业务需求定制自己所需的功能函数。

4. 自定义 Java 函数

在 Oracle 10g 以及以上版本中，支持使用 Java 语言编写的函数。对于习惯于使用 Java 语言的人，或者在已经通过 Java 语言编写了功能强大的类的环境下，可以采用下面这种方式定义 Java 函数。

创建包含要发布的公共静态方法的 Java 类。此时，需要创建一个类，该类中包含要发布的 Java 函数，并且要求该方法是公共（Public）的和静态（Static）的，定义如下 Java 类：

```
public class AreaJava{
  public static float areaj(float f){
     float area = 3.14*(f*f);
     return area;
  }
}
```

该类中包含一个公共的静态方法 area，保存该类为 AreaJava.java 文件，然后编译该类，如下所示。

```
F:\>javac AreaJava.java

F:\>
```

此时在F盘的根目录下创建了AreaJava.java的.class文件，接下来加载该文件到Oracle。

使用load.java把编译后的.class文件加载到Oracle，此时将该类加载到SCOTT用户模式。之前必须受以用户SCOTT JAVAUSERPRIV特权。授权过程如实例19-13所示。

【实例19-13】授以用户SCOTT的JAVAUSERPRIV特权。

```
F:\>sqlplus /nolog

SQL*Plus: Release 10.2.0.1.0 - Production on 星期二 8月 18 21:07:02 2009

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> conn system/oracle@orcl
已连接。
SQL> grant javauserpriv to scott;

授权成功。
```

然后把类AreaJava.class加载到SCOTT用户模式，如实例19-14所示。

【实例19-14】加载Java类文件到Oracle的SCOTT模式。

```
F:\>loadjava -verbose -schema scott -thin -user
scott/oracle@localhost:1521:orcl AreaJava.class
arguments: '-verbose' '-schema' 'scott' '-thin' '-user'
'scott/oracle@localhost:1521:orcl' 'AreaJava.class'
creating : class SCOTT.AreaJava
loading  : class SCOTT.AreaJava
Classes Loaded: 1
Resources Loaded: 0
Sources Loaded: 0
Published Interfaces: 0
Classes generated: 0
Classes skipped: 0
Synonyms Created: 0
Errors: 0
```

显示加载成功，整个加载过程只加载了一个类，没有错误发生。

为了调用该Java函数必须创建一个PL/SQL封装。此处建立一个PL/SQL函数，该函数封装了Java函数，如实例19-15所示。

【实例19-15】创建Java函数的PL/SQL封装。

```
SQL> create or replace function areajava(f float) return float
  2  as language java name 'AreaJava.areaj(double) return double';
  3  /

函数已创建。
```

下面验证该函数是否记录在数据字典中，查询数据字典 USER_OBJECTS 如实例 19-16 所示。

【实例 19-16】查询数据字典 USER_OBJECTS 验证 Java 函数。

```
SQL> col object name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type ='FUNCTION';

OBJECT_NAME          OBJECT_TYPE        CREATED       STATUS
----------------------------------- ---------------------------------
FTOC                  FUNCTION           15-8 月 -09    VALID
FTOCORACLE           FUNCTION           15-8 月 -09    VALID
AREA                  FUNCTION           18-8 月 -09    VALID
FINDWEALTHIER         FUNCTION           18-8 月 -09    VALID
AREAJAVA              FUNCTION           18-8 月 -09    VALID
MONYEMAKER           FUNCTION           18-8 月 -09    INVALID
MONEYMAKER           FUNCTION           18-8 月 -09    VALID

已选择 7 行。
```

在 OBJECT_NAME 列有函数 AREAJAVA，并且该行记录的 STATUS 列值为 VALID，说明该函数是有效的，可以供用户使用。下面调用自定义的 Java 函数 AREAJAVA，如实例 19-17 所示。

【实例 19-17】调用 Java 函数 AREAJAVA。

```
SQL> select areajava(8) from dual;

  AREAJAVA(8)
----------------------------
    200.96
```

19.3　游标

19.3.1　游标概述

在使用 SQL 的 SELECT 语句查询数据时，往往会返回一组记录的集合，为了依次处理记录集合中的每条记录，Oracle 使用游标来完成，遍历每个记录的功能。其实，游标可以看作指向记录集合的指针，它可以在集合记录中移动，以访问每条记录，如图 19-4 所示。

图 19-4 中，用户选择表 EMP 中的 JOB 为 MANAGER 的记录，而该查询结果有三条记录，通过使用 LOOP 循环语句，结合使用游标就可以遍历整个查询记录集合。

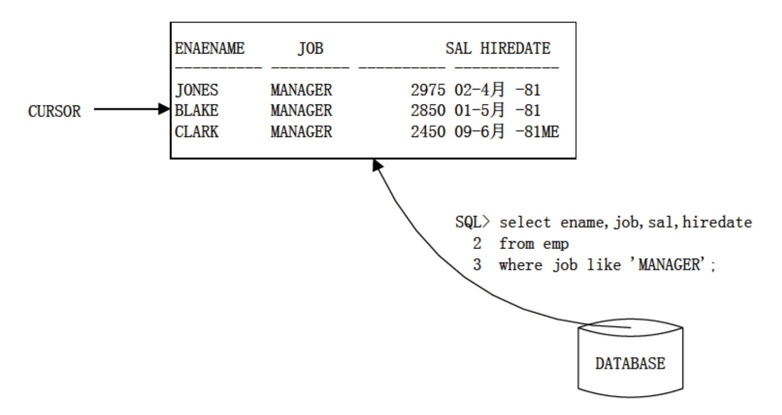


图 19-4 游标的作用示意图

1. 游标的创建

创建游标的语法如下所示：

```
CURSOR cursor_name IS sql_statements;
```

CURSOR 此时声明的是游标，而随后是用户自定义的游标的名字，在关键字 IS 之后是与游标相关联的 SQL 语句。

【实例 19-18】创建游标。

```
CURSOR cselectemp
IS
  select ename,job,sal
   from scott.emp;
```

此时创建了一个游标，游标名为 cselectemp，与该游标相关联的 SQL 语句为“select ename,job,sal from scott.emp;”，使用游标 cselectemp 就可以遍历查询结果中的记录集合。一旦游标创建成功，Oracle 就为其分配内存，并与定义的 SQL 语句关联起来。

2. 游标的打开

打开游标的语法如下：

```
OPEN cursor_name [argument [,argument……]];
```

打开游标的内部操作是 Oracle 执行与游标创建时相关联的 SQL 语句，OPEN 是关键字，随后是游标名，此时可以提供参数，这些参数用于传递为游标创建的语句的任何值。打开游标 cselectemp 的内部行为就是执行 SQL 语句。

3. 游标数据的读取

游标可以遍历整个记录集合，依次访问这些记录，其语法如下所示：

```
FETCH cursor_name INTO variable [,variable];
```

执行上述指令时，只能取记录集合中的一行记录，将这行记录放入随后的变量中，这些变量

的数据类型必须与记录中的每列的数据类型相同。

在实际中都是使用 LOOP 循环来实现记录集合的遍历，如使用 LOOP…EXIT WHEN 循环实现，如实例 19-19 所示。

【实例 19-19】游标取数据。

```
LOOP
 FETCH cselectemp
 INTO employeename,employeejob,employeesal;
 EXIT WHEN cselectemp%notfound;
DBMS_OUTPUT.put_line('employeename is '
                   ||employeename
                   ||'employeeejob is '
                   ||employeejob
                   ||'employeesal is'
                   ||employeesal );
END LOOP;
```

这里循环结束的条件是游标的属性值“%NOTFOUND”为真，即游标已经遍历完所有记录，每次循环将读取的记录信息输出显示。

4. 游标的关闭

关闭游标的语法如下：

```
CLOSE cursor_name.
```

一旦关闭游标，则 Oracle 释放为其分配的内存，游标不能重复打开，在打开前必须首先将其关闭。

5. 游标的属性

游标一旦打开，就处于某种状态中，为了了解游标的状态，以及使用这些状态实现某些逻辑操作，我们需要知道游标的属性，如下所示。

- %ISOPEN：判断游标是否是打开的。
- %FOUND：游标发现数据。
- %NOTFOUND：游标没有发现数据。
- %ROWCOUNT：游标可以遍历的记录的数量。

在使用游标的实例中给出一个实例来说明如何使用这些值确定游标的状态。访问游标属性的语法如下。

```
cursor_name.attribute_name
```

6. 游标的使用实例

下面通过一个具体的实例说明创建和使用游标的全过程，在该过程中使用游标来遍历查询的数据记录集合。下面这个实例采用步进的方式，不断完善整个过程的程序代码。

（1）声明变量

首先创建一个过程，该过程的声明如下：

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
```

过程名为cursortest，该过程声明了三个变量，即employeename、employeejob和employeesal，分别和用户SCOTT的表EMP的三个列，即ENAME、JOB和SAL对应。

（2）声明游标

接着，声明游标变量，即创建游标，代码如下所示：

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
```

这里声明游标在过程的声明后进行，该游标的名字为cselectemp，与之关联的SQL语句为“select ename,job,sal from scott.emp;”。此时Oracle为游标cselectemp分配内存。

（3）打开游标

在声明游标后，需要执行该游标以读取数据，所以接着执行打开游标操作，如下所示：

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
```

打开游标在过程的执行区进行，一旦打开游标，Oracle立即执行与之相关联的SQL语句。而后可以使用游标读取数据。

（4）读取数据

因为游标访问的数据是一个记录集合，所以往往需要一个循环语句来完成整个记录结合的访问。此时，使用一个LOOP…EXIT WHEN来循环访问数据记录集合。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
loop
 fetch cselectemp
  into employeename,employeejob,employeesal;
 exit when cselectemp%notfound;
 dbms_output.put_line('employeename is '
                     ||employeename
                     ||'employeeejob is '
                     ||employeejob
                     ||'employeesal is'
                     ||employeesal );
end loop;
end;
```

在 LOOP 循环中，如果游标移动过程中发现记录存在就继续执行循环，并每次把数据打印的标准输入输出装置。一旦游标移动后不能发现记录存在就结束循环，注意此时使用了游标的属性“%NOTFOUND”。但是读者或许注意到这里没有释放游标，游标是占用内存的，所以使用之后必须关闭游标以释放内存资源。

（5）释放资源

最后，关闭游标，完成整个包含游标的过程的创建，如实例 19-20 所示。

【实例 19-20】在存储过程中创建游标的完整示例。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
loop
 fetch cselectemp
  into employeename,employeejob,employeesal;
 exit when cselectemp%notfound;
 dbms_output.put_line('employeename is '
```

```
                   ||employeename
                   ||'employeeejob is '
                   ||employeejob
                   ||'employeesal is'
                   ||employeesal );
end loop;
close cselectemp;
end;
```

将该代码在记事本中编辑，并保存为.SQL 脚本文件，然后在 SQL*Plus 中的 SCOTT 用户下编译该脚本的创建过程 cselectemp，下面执行该过程查看结果。

（6）运行包含游标的存储过程

在 SCOTT 模式下创建过程 cursortest 后，执行该过程，该过程执行中使用游标遍历访问 EMP 表的数据记录集合，然后输入这些行记录，直到游标的“%NOTFOUND”值为真时，则结束对记录集合的遍历。执行存储过程如下所示。

【实例 19-21】执行存储过程 cursortest。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> set serveroutput on;
SQL> execute cursortest
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

19.3.2 FOR 游标

使用 FOR 游标可以简化实例 19-20 中的代码，使用 FOR 游标不需要声明变量，也不需要显式地打开游标和关闭游标，这些都是自动执行的。使用 FOR 游标的语法如下所示。

```
FOR record IN cursor_name
LOOP
```

```
 Logical statements……
END LOOP;
```

在 FOR 游标的语法中，record 为一条记录的集合，它自动定义为一个“%ROWTYPE”类型的变量，“%ROWTYPE”变量包含对应于记录中的多列变量，通过这个变量可以依次访问该记录中的每个列值。LOOP 循环将自动遍历游标所涉及的记录集合，循环开始时打开游标，而当循环结束时，游标自动关闭。下面给出一个实例改写实例 19-20 中创建的过程。

【实例 19-22】使用 FOR 游标的示例。

```
create or replace procedure forcursortest
is
cursor cselectemp
is
       select ename,job,sal
       from scott.emp;
begin
for emp_record in cselectemp
loop
 dbms output.put line('employeename is '
                        ||emp_record.ename
                        ||'employeeejob is '
                        ||emp_record.job
                        ||'employeesal is'
                        ||emp_record.sal );
end loop;
end;
```

同样将上述文件在记事本中编辑，保存在 F 盘的根目录下，命名为 forcursortest.sql 文件，然后编译该文件创建过程 forcursortest，如下所示。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> @f:\forcursortest.sql;
 2  /

过程已创建。
```

通过数据字典 USER_PROCEDURES 验证是否创建了过程 FORCURSORTEST，如实例 19-23 所示。

【实例 19-23】检验过程 FORCURSORTEST 是否创建成功。

```
SQL> select object_name
  2  from user_procedures
  3  where object_name like '%TEST';

OBJECT NAME
------------------------------
GOTOTEST
PROTEST
CURSORTEST
```

```
FORCURSORTEST
```

可见过程 FORCURSORTEST 已创建成功，下面执行该过程，并查看执行结果。

```
SQL> set serveroutput on
SQL> execute forcursortest
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

从过程 FORCURSORTEST 的输出结果可以看出，该过程和上例中创建的过程具有同样的功能。显然使用 FOR 游标极大地简化了编码，并减少了代码量。

19.3.3 隐式游标

隐式游标是没有声明的游标，如在 Java 语言中的无名内隐类的概念。没有显式地创建该游标，只是在 PL/SQL 代码块中执行 SQL 语句，这些 SQL 语句就是隐式游标，隐式游标也有属性，如“%ROWCOUNT”，即该 SQL 语句返回的记录数量，下面给出一个实例来说明如何使用隐式游标，以及隐式游标的属性。修改实例 19-20，使用隐式游标记录表 EMP 中不同的工作岗位的数量，并打印到标准输出装置，如实例 19-24 所示。

【实例 19-24】使用隐式游标的示例。

```
create or replace procedure hidencursortest
is
jobnumber NUMBER;
cursor cselectemp
is
     select ename,job,sal
     from scott.emp;
begin
select count(distinct(job))
into jobnumber
from emp;
dbms_output.put_line('there are '
                   ||jobnumber
                   ||'diferent jobs');
```

```
    dbms_output.put_line('hiden cursor rowcount is '
                       ||SQL%ROWCOUNT);
    for emp record in cselectemp
    loop
     dbms_output.put_line('employeename is '
                       ||emp_record.ename
                       ||'employeeejob is '
                       ||emp_record.job
                       ||'employeesal is'
                       ||emp_record.sal );
    end loop;
    end;
```

在上例中，在过程的执行区开始 begin 后，使用了 SQL 语句，查询表 EMP 中不同工作岗位的数量，保存到过程中声明的变量 jobnumber 中，然后打印到输出装置。此处的 SQL 语句就是一个隐式游标，接着输出隐式游标执行的 SQL 语句返回的记录数，这说明了隐式游标的存在。在记事本中编辑该文件，并保存在 F 盘的根目录下，保存的文件名为 hidencursortest.sql。下面编译并执行该过程，如实例 19-25 所示。

【实例 19-25】编译脚本文件 hidencursortest.sql 创建过程 hidencursortest。

```
SQL>  @f:\hidencursortest.sql
 27  /

过程已创建。
```

下面执行过程 hidencursortest，观察输出结果，如实例 19-26 所示。

【实例 19-26】执行过程 hidencursortest。

```
SQL> execute hidencursortest
there are 5diferent jobs
hiden cursor rowcount is 1
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

在该过程中看到隐式游标计算了表 EMP 中不同岗位的数量，并且输出了隐式游标中的记录

行数量，该值为 1，符合隐式游标执行的 SQL 语句的返回结果，即函数 COUNT(DISTINCT(JOB))返回一行记录。

19.3.4 REF 游标

REF 游标，即引用游标，它是一个记录的集合，Oracle 允许在不同的程序单元之间传递游标的引用。

可以在存储过程中定义 REF 游标，这样可以在需要的时候使用引用游标，不必在需要游标时都显式定义。要使用 REF 游标，首先需要声明它为一个 TYPE，声明方式如下所示：

```
TYPE ref 游标名 IS REF CURSOR [返回类型]
```

如 TYPE ty_emprefcur IS REF CURSOR，说明 ty_emprefcur 是 REF 游标。在使用时需要创建该类型的一个实例，如下所示：

```
empcursor  ty_emprefcur;
```

此时 empcursor 就是 REF 游标 ty_emprefcur 的一个实例。当然还可以创建其他实例。下面给出一个完整的实例。

【实例 19-27】使用 REF 游标的示例。

```
create or replace procedure refcursortest
is
    type ty_emprefcur is ref cursor; --声明 REF 游标为一个 TYPE
    empcursor       ty_emprefcur; --创建该类型的一个实例
    rec_emp          emp%rowtype;
    rec_sal         emp.sal%type;
    rec_job          emp.job%type;
begin
  open empcursor for --使用 REF 游标
       select *
         from emp;

  fetch empcursor
        into rec_emp;
  close empcursor;
dbms_output.put_line('employeename is '||rec_emp.ename);

open empcursor for --使用 REF 游标
      select sal
          from emp;
fetch empcursor
   into rec_sal;
dbms_output.put_line('employee sal is '||rec_sal);
close empcursor;

open empcursor for --使用 REF 游标
      select job
         from emp;
```

```
 fetch empcursor
   into rec_job;
dbms output.put line('employee job is '||rec job);
end;
/
```

在 Windows 的记事本中编辑该文件，并保存在 F 盘的根目录下，文件名为 refcursortest.sql，然后创建这个存储过程，如实例 19-28 所示。

【实例 19-28】创建包含 REF 游标的存储过程。

```
SQL> @f:\refcursortest.sql;

过程已创建。
```

当存储过程创建成功之后，执行该过程，并查看输出结果，如实例 19-29 所示。

【实例 19-29】执行包含 REF 游标的存储过程 refcursortest。

```
SQL> execute refcursortest;
employeename is SMITH
employee sal is 800
employee job is CLERK

PL/SQL 过程已成功完成。
```

在过程 refcursortest 中，有三次使用 REF 游标，其实，在打开 REF 游标时就给出了与之关联的 SQL 语句，然后讲执行游标时返回的记录集合的第一个记录放入一个变量，然后输入该值。

如果希望 REF 游标返回的数据类型更严格，可以使用“%ROWTYPE”，如下所示。

```
TYPE empcursor IS REF CURSOR
  RETURN emp%rowtype
```

19.4　本章小结

本章介绍了三个数据库对象，即存储过程、函数和游标。存储过程是一段执行特殊功能的 PL/SQL 代码，它保存在数据库端，存储过程可以有参数，但是没有返回值。而函数也是一段 PL/SQL 代码，函数可以有不止一个参数，也可以不使用任何参数，但是它必须有返回值，在存储过程中可以显式地调用用户创建的函数和 Oracle 设计的系统函数。

在用户读取一组记录集合时，往往需要依次访问记录集合中的每条记录，使用游标可以轻松实现该功能。游标占有内存且和指定的 SQL 语句关联，如果不需要游标时，必须关闭游标以释放内存资源。游标不仅仅有显式游标，还有 FOR 游标、隐式游标和 REF 游标，掌握不同类型的游标，有助于编写简洁的过程代码，或者节省系统资源。

第 20 章

◀触　发　器▶

触发器类似于 Oracle 的 PL/SQL 存储过程，但是它不能被显式调用，而是由数据库服务器维护，在特定事件发生时由 Oracle 数据库调用，触发器使用 PL/SQL 语言编写保存在数据库服务器上。Oracle 提供了在 DML 操作和 DDL 操作之前和之后的触发器，还提供了当数据库关闭与启动或者用户登录与退出的触发器，这些触发器极大地丰富了 DBA 执行、审计、安全或完整性关联的管理任务。

20.1　触发器的创建

本节只给出一个创建触发器的实例，通过这个实例读者可以直观感受创建触发器的语法，以及如何创建一个标准的 DML 触发器。

创建触发器的语法为：

```
CREATE TRIGGER trigger name
BEFORE/AFTER DELETE [UPDATE,INSERT,SHUTDOWN……]
ON object_name
Trigger_SQL_PL/SQL_body
```

其中，CREATE TRIGGER 表示要创建一个触发器，随后是触发器名→触发触发器的时机，即 BEFORE 或 AFTER→触发事件如 DML 操作 DELETE 或数据库关闭操作 SHUTDOWN。使用 ON 关键字说明触发器操作的对象，该对象可以是表或者数据库（DATABASE）。最后是触发器的主体代码逻辑，这里详细说明激发触发器后的数据库行为。

下面创建一个触发器，该触发器实现对 SCOTT 模式下的表 EMP 的操作记录，即 DML 操作的记录，当发生任何 DML 操作时，都通过标准输出装置打印一条信息。当然这里可以是更复杂的记录，如记录用户删除的数据、记录更新前的原始值、插入数据之前的原始值、记录用户 DML 操作的次数等，总之读者可以根据自己的需求进行设置。

【实例 20-1】创建一个标准的简单触发器。

```
create or replace trigger delete_emp_trigger
 before delete
```

```
        on emp
    for each row
       begin
        dbms_output.put_line('deleting......');
   end delete_emp_trigger;
   /
```

将上述代码在记事本中编译为一个.SQL 脚本文件，而在 SCOTT 模式下编译并创建触发器 delete_emp_trigger。笔者将该文件保存在 F 盘的根目录下，编译过程如下所示。

【实例 20-2】执行触发器脚本文件并创建触发器。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> @f:\delete emp trigger.sql;

触发器已创建
```

为了验证创建触发器的结果，可使用数据字典 USER_OBJECTS，如实例 20-3 所示。

【实例 20-3】使用数据字典 USER_OBJECTS 查询创建的触发器信息。

```
SQL> col object_name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3* where object type ='TRIGGER'

OBJECT_NAME         OBJECT_TYPE           CREATED     STATUS
------------------------------------ ---------------------------------
SCOTT_EMP_DML       TRIGGER               19-8 月 -09     VALID
DELETE_EMP_TRIGGER   TRIGGER               19-8 月 -09     VALID
```

可见触发器 DELETE_EMP_TRIGGER 创建成功，且该触发器为有效的触发器，因为该行的 STATUS 列值为 VALID。此时，触发器如存储过程和函数一样，它存储在表中，与具体的表相关联，在第一次被调用时即时编译。

下面我们演示触发器的行为、Oracle 数据库何时使用触发器。再强调一下，我们创建的触发器 DELETE_EMP_TRIGGER 的作用是在表 EMP 中删除行记录时，对于删除的每一行记录前都触发一个事件，该事件将向标准输出打印一行信息，其实，在现实的生产数据库中，触发器行为逻辑可能会很复杂，这里我们只是给出一个演示，希望读者通过这个简单的触发器行为，理解如何创建触发器和触发器的行为。

【实例 20-4】通过删除表 EMP 中的记录来验证触发器的行为。

```
SQL> delete from emp
  2  where ename ='FORD';
deleting.........

已删除 1 行。
```

从上述输出可以看出，当删除表 EMP 中的记录之前，触发器执行触发事件向标准输出打印

一行提示，然后删除满足删除条件的行记录。

20.2 触发器的分类及语法格式

Oracle 的数据库触发器可以对不同操作类型实现某种行为，如审计或安全考虑等，这些操作包括 DML 操作、DDL 操作和数据库级操作。

我们基于不同的操作类型分为三类。

1. 基于 DML 操作的触发器

触发器可以在当用户对一个表的 INSERT、UPDATE 和 DELETE 操作时触发行为，也可以实现对表的每一行进行 DML 操作时，实现触发器行为，此时需要在触发器的定义中使用 FOR EACH ROW 语句说明操作的每一行都触发某种触发器行为。

创建该类触发器的语法格式如下所示：

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[INSERT|UPDATE|DELETE] ON table_name [FOR EACH ROW [WHEN cond]]
Trigger_SQL_Statements;
```

创建触发器时 OR REPLACE 语句可以使用，也可以不使用，使用它的目的是如果当前用户模式下已经创建了该触发器则覆盖原触发器，BEFORE|AFTER 说明触发器被触发的时机，而 INSERT|UPDATE|DELETE 说明触发器被触发的事件，如 BEFORE INSERT 说明，在向表中插入数据前激发触发器行为，ON table_name 说明触发器作用的表名，FOR EACH ROW 说明触发器对操作如 INSERT 涉及的每一行都激发触发器行为，这样的触发器称为行级触发器，WHEN cond 给出更具体的条件，当满足某种条件时，再执行触发器行为。

2. 基于 DDL 操作的触发器。

DDL 操作，如 CREATE、ALTER 和 DROP，在执行这些操作前或者之后实现触发器行为，如用户删除了一个表，此时需要一个触发器来记录该用户删除的表的信息和该用户名作为用户操作日志，这也是此类触发器的典型应用。

创建该类触发器的语法格式如下所示：

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[CREATE|ALTER|DROP] ON database_name [WHEN cond]]
Trigger_SQL_Statements;
```

此类触发器在数据库中创建和删除数据库对象或者执行 ALTER 指令时激发触发器行为。

3. 基于数据库级操作的触发器

数据库级操作是指 STARTUP、SHUTDOWN、LOGON、LOGOFF 等数据库相关的操作，如用户登录时记录该用户登录时间和用户名，而当用户退出时，也记录该用户的退出时间等。对于数据库启动 STARTUP 和数据库关闭 SHUTDOWN 等同样可以编写符合业务需求的触发器。

创建该类触发器的语法格式如下所示：

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[START|SHUTDOWN|LOGON|LOGOFF] ON database_name [WHEN cond]]
Trigger_SQL_Statements;
```

此类触发器在数据库级行为，如关闭和启动数据库、用户登录和退出数据库时激发触发器行为。按照触发器操作对象的粒度不同，也可以分为语句级触发器和行级触发器，但是使用操作类型分类更直观。

20.3　触发器的创建权限

在用户创建触发器时，必须具有 CREATE TRIGGER 权限，如果用户不具备这个权限，则需要在 DBA 用户下，使用 GRANT CREATE TRIGGER TO user_name 指令赋予当前用户权限 ，而如果需要在当前用户下，创建其他用户的触发器，则需要具有 CREATE ANY TRIGGER 的权限，这里 ANY 的含义就是为任何用户创建触发器。如果要创建的触发器是作用是在数据库上，如对 STARTUP 或 SHUTDOWN 事件激发触发器，则需要具有 ADMINISTER DATABASE TRIGGER 的系统权限。

如在 SCOTT 模式下，可以通过如下方式查看当前用户是否有创建触发器的权限。

【实例 20-5】查看当前用户的系统权限。

```
SQL> conn system/oracle@orcl
已连接。
SQL> select *
  2  from dba_sys_privs
  3  where grantee = 'SCOTT';

GRANTEE                      PRIVILEGE                     ADM
------------------------------------------------ -------------
SCOTT                         UNLIMITED TABLESPACE          NO
SCOTT                         CREATE TRIGGER                NO
```

从输出可以看出该 SCOTT 用户具有创建触发器的权利，但是只能创建自己模式下的触发器，如果 PRIVILEGE 为 CREATE ANY TRIGGER，则可以创建任何模式下的触发器。可以通过如下授权实现，如实例 20-6 所示。

【实例 20-6】授权 SCOTT 用户 CREATE ANY TRIGGER 的权限。

```
SQL> GRANT create any trigger to SCOTT;

授权成功。
```

此时，显示授权成功，为了验证用户 SCOTT 的权限信息，再次使用数据字典 DBA_SYS_PRIVES 来查看用户 SCOTT 的系统权限，如实例 20-7 所示。

【实例 20-7】查看用户是否具有 CREATE ANY TRIGGER 的权限。

```
SQL> col grantee for a15
SQL> col privilege for a20
SQL> col admin_option for a15
SQL> select *
  2  from dba_sys_privs
  3  where grantee = 'SCOTT'
  4* and privilege like 'CREATE%'

GRANTEE            PRIVILEGE              ADMIN_OPTION
-------------------------- --------------------------------- ---

SCOTT              CREATE TRIGGER         NO
SCOTT              CREATE ANY TRIGGER     NO
```

此时，数据字典 DBA_SYS_PRIVS 中记录了用户 SCOTT 具有 CREATE ANY TRIGGER 的权限。



ADMIN_OPTION 选项的含义是 SCOTT 用户具有的权利是否可以再赋予其他用户，如果 ADMIN_OPTION 为 NO，则说明对应的权利不能再赋予其他用户，如果 ADMIN_OPTION 为 YES，则说明对应的权限继续赋予其他用户。

在创建基于 DML 操作的触发器时，由于操作的是表对象，所以有一个可选项，即 FOR EACH ROW，以实现对每一行都激发触发器行为。此时 Oracle 提供了两个临时表来访问每行中的新值和旧值，即:new 和:old。下面给出一个实例说明它们的作用。

编写一个触发器，作用是更新 SCOTT 用户的表 EMP 中的记录后，在将更新的行记录对象的工资的旧值和新值打印到标准输出装置，如实例 20-8 所示。

【实例 20-8】创建表 EMP 的 UPDATE 触发器。

```
create or replace trigger update_emp_trigger
 after update on emp
 for each row
begin
dbms_output.put_line('old value is '||:old.sal);
dbms_output.put_line('new value is'||:new.sal);
end update_emp_trigger;
```

将该文件保存在 F 盘根目录下，名字为 updateemp.sql，然后执行该脚本文件创建触发器 update_emp_trigger，如实例 20-9 所示。

【实例 20-9】执行脚本文件创建触发器 UPDATE_EMP_TRIGGER。

```
SQL> @f:\updateemp;
  8  /

触发器已创建
```

此时，已经成功创建触发器 UPDATE_EMP_TRIGGER，通过数据字典查看该触发器的信息，

如实例 20-10 所示。

【实例 20-10】通过数据字典 USER_OBJECTS 查看触发器信息。

```
SQL> col object name for a25
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type = 'TRIGGER'
  4* and status = 'VALID'

OBJECT_NAME                OBJECT_TYPE          CREATED        STATUS
------------------------------------------- ------------------
CHECK_EMP_SALARY            TRIGGER              20-8 月 -09     VALID
UPDATE_EMP_TRIGGER          TRIGGER              19-8 月 -09     VALID
```

可见触发器 UPDATE_EMP_TRIGGER 创建成功并记录在数据字典中，该触发器是有效的触发器（STATUS 为 VALID）。下面更新表 EMP 中的数据，观察触发器的行为，如实例 20-11 所示。

【实例 20-11】更新表 EMP 中的数据。

```
SQL> set serveroutput on
SQL> update emp
  2  set sal = sal+200
  3  where job ='CLERK';
old value is 800
new value is 1000
old value is 1100
new value is 1300
old value is 950
new value is 1150
old value is 1300
new value is 1500

已更新 4 行。
```

从输出可以看出，在表 EMP 上更新相关列 SAL 的值之后，触发器开始工作，先输出每一行中 SAL 的旧值，在输出每一行中更新后的 SAL 新值，因为更新语句是将 EMP 表中岗位为 CLERK 的员工工资统一增加 200，所以每行的旧值总是比新值少 200。因为更新了 4 行记录，所以触发器的输出有 8 行，每行涉及一对新值和旧值。

20.4　触发器的审核

本节将给出一个具有实际意义的触发器，即审核触发器，当用户操作一个重要的表时，如插入数据和更新数据，我们希望记录该用户的名称和更改时间等信息，以备将来审核使用。

在创建审核触发器前，需要先创建一个表，该表用于存储审核信息，如实例 20-12 所示。

【实例 20-12】创建审核表。

```
SQL> create table user_modify_table
  2  ( username  varchar2(20),
  3    modifytime   date,
  4    content   varchar2(20));

表已创建。
```

成功创建表 USER_MODIFY_TABLE 后，使用 DESC 指令查看该表的结构信息，如实例 20-13 所示。

【实例 20-13】查看表 USER_MODIFY_TABLE 的结构信息。

```
SQL> desc user_modify_table;
 名称                                              是否为空? 类型
 ----------------------------------------------------------------- ----
 USERNAME                                                   VARCHAR2(20)
 MODIFYTIME                                                 DATE
 CONTENT                                                    VARCHAR2(20)
```

此时，使用表 USER_MODIFY_TABLE 存储用户的信息，下面创建一个触发器。当用户对表 EMP 执行 UPDATE 或 INSERT 操作时，记录用户的审核信息。

创建审核触发器，当发生对表 EMP 的 INSERT 操作和 UPDAT 操作时，将用户名、操作日期和操作内容记录到表 USER_MODIFY_TABLE 中，如实例 20-14 所示。

【实例 20-14】创建审核触发器。

```
CREATE OR REPLACE TRIGGER user_change_empdata
BEFORE update or insert ON emp
FOR EACH ROW
BEGIN
    IF inserting THEN
INSERT INTO user modify table
 VALUES (user,sysdate,'inserting');
END IF;
IF updating THEN
INSERT INTO user modify table
 VALUES (user,sysdate,'updating');
END IF;
END user_change_empdata;
/
```

将上述代码在记事本中编辑并保存在 F 盘的根目录下，文件名为 user_change_empdata.sql。下面执行该脚本文件，创建触发器，如实例 20-15 所示。

【实例 20-15】执行 user_change_empdata.sql 脚本文件创建触发器。

```
SQL> @f:\user_change_empdata.sql;

触发器已创建
```

同样查询该触发器在数据字典 USER_OBJECTS 中的信息，如实例 20-16 所示。

【实例 20-16】查询触发器 user_change_empdata 的信息。

```
SQL> select object_name,object_id,object_type,created,status
  2  from user_objects
  3  where object_type = 'TRIGGER'
  4  and object_name like 'USER%';

OBJECT_NAME            OBJECT_ID OBJECT_TYPE   CREATED    STATUS
-------------------------------------------- --------------- ---------------
USER_CHANGE_EMPDATA  53299     TRIGGER        20-8 月 -09   VALID
```

可见已经成功创建了触发器 USER_CHANGE_EMPDATA。而且该触发器为有效（VALID）的，所以当对表 EMP 进行 UPDATE 或 INSERT 操作时，会激发触发器，将用户操作信息记录到表 USER_MODIFY_TABLE 中。进行 INSERT 和 UPDATE 操作，如实例 20-17 所示。

【实例 20-17】对表 EMP 进行 INSERT 和 UPDATE 操作。

```
SQL> insert into emp (empno,ename,sal,job)
  2  values (7899,'tom',10000,'MANAGER');

已创建 1 行。
SQL> update emp
  2  set sal = sal +200
  3  where job ='CLERK';

已更新 4 行。
```

为了验证触发器是否被触发，可以查看表 USER_MODIFY_TABLE 的内容，如实例 20-18 所示。

【实例 20-18】查看表 USER_MODIFY_TABLE。

```
SQL> SELECT *
  2  FROM USER_MODIFY_TABLE;

USERNAME          MODIFYTIME     CONTENT
-------------------------------- ---------------
SCOTT               20-8 月 -09      inserting
SCOTT               20-8 月 -09      updating
SCOTT               20-8 月 -09      updating
SCOTT               20-8 月 -09      updating
SCOTT               20-8 月 -09      updating
```

可见，对表 EMP 的一行进行了 INSERT 操作和对表中的 4 行进行了 UPDATE 操作。从表 USER_MODIFY_TABLE 的记录可以知道触发器 USER_CHANGE_EMPDATA 被成功触发。

20.5 触发器的删除

删除触发器的作用是对表中的数据进行删除 DELETE 操作时，记录删除的内容，在某些场合是处于备份的需要，同样使用表 EMP 作为删除触发器的操作对象。

为了保存删除的数据，需要创建一个记录删除数据的表，该表的结构与 EMP 表的结构对应，即列的数据类型和数量都相等，如实例 20-19 所示。

【实例 20-19】创建记录删除数据的表 BACKUP_DELETE_EMP_TABLE。

```
SQL> create table backup_delete_emp_table
  2  (backempno  number(4),
  3  backename   varchar2(10),
  4  backjob     varchar2(9),
  5  backmgr     number(4),
  6  backhiredate date,
  7  backsal     number(7,2),
  8  backcomm    number(7,2),
  9  backdeptno number(2));

表已创建。
```

下面开始创建一个删除触发器，备份删除的表 EMP 中的记录，如实例 20-20 所示。

【实例 20-20】创建删除触发器定义。

```
create or replace trigger backup_emp_trigger
before delete on emp
for each row
begin
insert into backup_delete_emp_table
values(:old.empno,:old.ename,:old.job,:old.mgr,:old.hiredate,
      :old.sal,:old.comm,:old.deptno);
end backup_emp_trigger;
```

将上述文件在记事本中编辑，命名为 backup_emp_trigger.sql，并保存在 F 盘根目录下，下面执行脚本文件创建触发器 backup_emp_trigger.sql，如实例 20-21 所示。

【实例 20-21】创建触发器 backup_emp_trigger。

```
SQL> @f:\backup_emp_trigger.sql;
  9  /

触发器已创建
```

输出提示触发器创建成功，为了演示触发器工作的结果，先在 EMP 表上执行两个删除操作，如实例 20-22 所示。

【实例 20-22】在 EMP 表上执行删除操作。

```
SQL> delete from emp
```

```
  2  where ename = 'tom';

已删除 1 行。
SQL> delete from emp
  2  where ename = 'SCOTT';

已删除 1 行。
```

因为触发器 backup_emp_trigger 是作用在表 EMP 上，并且当从表 EMP 中删除数据记录时，激发该触发器，使得被删除的记录保存在表 BACKUP_DELETE_EMP_TABLE 中。下面查询表 BACKUP_DELETE_EMP_TABLE 的内容，如实例 20-23 所示。

【实例 20-23】查询表 BACKUP_DELETE_EMP_TABLE 的内容。

```
SQL> select backempno,backename,backjob,backhiredate,backsal
  2  from backup_delete_emp_table;

 BACKEMPNO BACKENAME  BACKJOB   BACKHIREDATE      BACKSAL
-------------------- -------------------- -------------- -----------
     7899     tom           MANAGER                        10000
     7788    SCOTT       ANALYST    19-4 月 -87             3000
```

我们看到员工 tom 和 SCOTT 正是刚刚删除的记录，而他们的记录信息保存在表 BACKUP_DELETE_EMP_TABLE 中，说明触发器 backup_emp_trigger 被 DELETE 事件成功激发。

20.6 触发器定义中的条件语句

在触发器中为了更加细粒度的控制触发器激发条件，允许使用条件语句，主要有两类条件语句，一个是 WHEN 子句，另一个 IF 子句。本节依次讲解这两类条件语句的用法，并通过实例使得读者理解如何运用这两类条件语句。

20.6.1 WHEN 条件语句

对于创建触发器，往往需要进行更详细的条件控制，使用 WHEN 子句可以设置触发器基于行级的触发器条件，即对表的每一行的操作进行更加细致的条件判断，从而执行某些行为。在触发器中使用 WHEN 子句的语句结构如下所示：

```
CREATE OR REPLACE TRIGGER when_emp_trigger
BEFORE UPDATE OR INSERT ON emp
FOR EACH ROW
WHEN (condition)
BEGIN
Trigger body;
END when_emp_trigger;
```

更改删除触发器的实例，在删除表 EMP 的一行记录时，先判断要删除的记录的岗位，即 JOB，

如果岗位是 MANAGER 或者 PRESIDENT，则备份删除的记录，而删除的其他记录不需要做备份。修改后的删除触发器如实例 20-24 所示。

【实例 20-24】创建使用 WHEN 语句的删除触发器。

```
CREATE OR REPLACE TRIGGER when_backup_emp_trigger
BEFORE delete ON emp
FOR EACH ROW
WHEN (old.job IN 'MANAGER,PRESIDENT')
BEGIN
INSERT INTO backup_delete_emp_table
VALUES(:old.empno,:old.ename,:old.job,:old.mgr,:old.hiredate,
      :old.sal,:old.comm,:old.deptno);
END when_backup_emp_trigger;
```

此时在 FOR EACH ROW 后使用了 WHEN 条件语句，这样使得触发器触发的时机条件更加详细，即实现更加细粒度的条件控制。



在使用 WHEN 子句时，WHEN 子句中的 new 或者 old 不使用:old 或:new 的形式，并且在 WHEN 子句后没有分号。

20.6.2 IF 条件语句

在对表创建触发器时，有几个 DML 操作来激发触发器行为，这些操作包括 INSERT、UPDATE 和 DELETE，一个触发器可以响应多个触发事件而执行不同的行为，如被 INSERT 事件触发时，希望保存旧的行记录，并保存该操作事件，而使用 DELETE 事件触发时，则需要记录用户名和 DELETE 事件，此时就需要一个条件判断。

Oracle 设计了三个布尔表达式来表示三个不同的触发事件，即 INSERTING、UPDATING 和 DELETING。其使用方式如下所示：

```
IF [INSERT | UPDATE | DELETE ] THEN
  trigger body;
END IF ;
```

20.7 基于 Java 语言的触发器

Oracle 允许在创建触发器时，使用 Java 语言编写的代码执行触发器的执行部分，但是需要读者注意，创建基于 Java 语言的触发器不是直接将 Java 代码关联为触发器，而是在触发器的执行部分调用 Java 代码。下面创建一个包含 Java 代码的触发器，它的作用是当对 EMP 表执行删除操作前，将用户名、删除时间和删除行为保存到表 USER_MODIFY_TABLE 中。首先创建一个 Java 类 delemptrigger，如实例 20-25 所示。

【实例 20-25】创建 Java 类 DelempTrigger。

```
import java.io.*;
import java.sql.*;
import oracle.sql.*
import oracle.core.lmx.*

public class DelempTrigger{

public static void recordDeleteData() throws SQLException,CoreException
 {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stm = conn.CreateStatement();
   stm.execute("INSERT INTO user_modify_table
              VALUES (user,sysdate,'deleting') ");
   stm.close();
   return;
 }
}
```

该类定义了一个 PUBLIC STATIC 方法 recordDeleteData()，它是 Java 代码执行触发器主体行为的方法，然后编译并生成 DelempTrigger.class 文件。



在编译该 DelempTrigger.java 文件前需要在环境变量中将 Oracle 自带的一个 class12.jar 文件包含在内，Oracle 10g 中目录为 F:\oracle\product\10.2.0\db_1\jdbc\lib。

最后创建一个过程用于包装 Java 代码，如实例 20-26 所示。

【实例 20-26】创建包含 Java 代码的存储过程 record_deleteemp_user_trigger。

```
Create or replace procedure record_deleteemp_user_trigger( )
Is language java
Name 'DelempTrigger. recordDeleteData( )';
```

最后定义一个存储过程，在过程的执行部分调用封装了 Java 代码的存储过程，如实例 20-27 所示。

【实例 20-27】创建基于 Java 语言的存储过程 user_change_empdata。

```
CREATE OR REPLACE TRIGGER user_change_empdata
BEFORE delete ON emp
FOR EACH ROW
BEGIN
--调用封装了 JAVA 代码的存储过程
 call record deleteemp user trigger();
END user_change_empdata;
```

在该存储过程的执行部分调用了另一个存储过程 record_deleteemp_user_trigger，而该过程是对 Java 代码的封装。

20.8 触发器的管理

触发器管理是触发器维护的重要内容，首先在维护前需要知道触发器的信息，Oracle 提供了 USER_TRIGGERS 数据字典查看当前用户的触发器信息。触发器依赖性失效后可以重新编译该触发器，如果临时不需要执行触发器可以屏蔽掉，如果永久不需要该触发器可以删除触发器。

20.8.1 查看触发器

本章已经建立了很多触发器，都是对表 EMP 的事件触发行为，那么如何查看创建的触发器的信息呢？Oracle 提供了数据字典 USER_TRIGGERS。通过它可以查看触发器的所有信息，首先通过实例 20-28 查看数据字典的结构。

【实例 20-28】查看数据字典 USER_TRIGGERS 的结构。

```
SQL> desc user_triggers;
 名称                                        是否为空? 类型
 ----------------------------------------------------- --
 TRIGGER_NAME                                VARCHAR2(30)
 TRIGGER_TYPE                                VARCHAR2(16)
 TRIGGERING_EVENT                            VARCHAR2(227)
 TABLE_OWNER                                 VARCHAR2(30)
 BASE_OBJECT_TYPE                            VARCHAR2(16)
 TABLE_NAME                                  VARCHAR2(30)
 COLUMN_NAME                                 VARCHAR2(4000)
REFERENCING_NAMES                            VARCHAR2(128)
WHEN_CLAUSE                                  VARCHAR2(4000)
 STATUS                                      VARCHAR2(8)
 DESCRIPTION                                 VARCHAR2(4000)
 ACTION_TYPE                                 VARCHAR2(11)
 TRIGGER_BODY                                LONG
```

下面依次说明这些参数的作用，这样读者就可以通过数据字典 USER_TRIGGERS 查询需要的当前用户的触发器信息。

- TRIGGER_NAME：触发器名称，如 BACKUP_EMP_TRIGGER。
- TRIGGER_TYPE：触发器类型，说明是行级触发器还是语句级触发器，如果是行级触发器，该值如 BEFORE EACH ROW 或 AFTER EACH ROW。
- TRIGGERING_EVENT：触发事件，如 DELETE、UPDATE、INSERT 等。
- TABLE_OWNER：触发器所关联的表的拥有者。
- BASE_OBJECT_TYPE：触发器所关联的是表 TABLE，还是数据库 DATABASE。
- TABLE_NAME：触发器所关联的表名。
- COLUMN_NAME：触发器所关联的列名，Oracle 允许更细粒度的触发器激发控制，可以在用户操作表的某一列时执行触发器行为。
- WHEN_CLAUSE：触发器中的 WHEN 条件语句内容。

- STATUS：说明当前的触发器是否被屏蔽，DISABLE 表示被屏蔽了，即没有激活该触发器，ENABLE 表示激活了该触发器，可以使用。
- DESCRIPTION：描述说明触发器名、触发事件和时机以及关联的对象和触发器类型（行级触发器还是语句级触发器），以下是一个 DESCRIPTION 的实例：

```
backup_emp_trigger
before delete on emp
for each row
```

- TRIGGER_BODY：触发器执行部分，以下是触发器 BACKUP_EMP_TRIGGER 的 TRIGGER_BODY 的内容：

```
begin
insert into backup_delete_emp_
table
values(:old.empno,:old.ename,:
old.job,
```

下面，给出一个实例查看触发器 BACKUP_EMP_TRIGGER 的信息。

【实例 20-29】查看触发器 BACKUP_EMP_TRIGGER 的信息。

```
SQL> set line 120
SQL> col status for a15
SQL> col triggering_event for a15
SQL> col description for a25
SQL>select trigger_name,triggering_event,description,status
  2  from user_triggers
  3* where trigger_name = 'BACKUP_EMP_TRIGGER'

TRIGGER_NAME        TRIGGERING_EVEN DESCRIPTION          STATUS
------------------------------------ -------------------------- -------
BACKUP_EMP_TRIGGER   DELETE          backup_emp_trigger     ENABLED
                                      before delete on emp
                                      for each row
```

可见触发器 BACKUP_EMP_TRIGGER 的触发事件是 DELETE 操作，作用的表为 EMP，当前的状态为激活状态，因为 STATUS 为 ENABLED。

20.8.2　重编译触发器

当触发器被标记为无效时，此时它不能被执行，需要重新编译该触发器，手工编译触发器的指令如下所示：

```
ALTER TRIGGER trigger_name COMPILE;
```

下面给出一个实例，说明如何重新编译触发器 backup_emp_trigger。

【实例 20-30】手工编译触发器 backup_emp_trigger。

```
SQL> alter trigger backup_emp_trigger compile;
```

```
触发器已更改
```

20.8.3 屏蔽触发器

如果一个触发器不需要执行，但是又不希望删除它，则可以屏蔽该触发器，屏蔽触发器使用指令 ALTER TRIGGER trigger_name DISABLE，如实例 20-31 所示。

【实例 20-31】屏蔽触发器 BACKUP_EMP_TRIGGER。

```
SQL> alter trigger backup_emp_trigger disable;

触发器已更改
```

在屏蔽掉触发器后，通过实例查看触发器 BACKUP_EMP_TRIGGER 的状态，如实例 20-32 所示。

【实例 20-32】查看触发器 BACKUP_EMP_TRIGGER 的状态。

```
SQL> col table_name for a10
SQL>select trigger_name,triggering_event,table_name,status
  2  from user triggers
  3* where trigger_name ='BACKUP_EMP_TRIGGER'

TRIGGER_NAME         TRIGGERING_EVEN TABLE_NAME STATUS
------------------------------------ ---------------------------- ----
BACKUP_EMP_TRIGGER   DELETE             EMP        DISABLED
```

可见触发器 BACKUP_EMP_TRIGGER 已经被屏蔽了，STATUS 值为 DISABLED。那么如何激活该触发器呢？启动触发器时可如下语句实现：

```
ALTER TRIGGER trigger_name ENABLE;
```

【实例 20-33】开启触发器 BACKUP_EMP_TRIGGER。

```
SQL> alter trigger backup emp trigger enable;

触发器已更改
```

通过数据字典 USER_TRIGGERS 查看触发器 BACKUP_EMP_TRIGGER 的状态，如实例 20-34 所示。

【实例 20-34】查看开启后的触发器 BACKUP_EMP_TRIGGER 状态。

```
SQL> select trigger name,triggering event,table name,status
  2  from user_triggers
  3  where trigger_name ='BACKUP_EMP_TRIGGER';

TRIGGER NAME         TRIGGERING EVEN TABLE NAME STATUS
------------------------------------ ------------------------
BACKUP_EMP_TRIGGER   DELETE           EMP        ENABLED
```

注意此时触发器 BACKUP_EMP_TRIGGER 的 STATUS 为 ENABLED，说明该触发器被重新启动。

20.8.4 删除触发器

当不需要一个已经创建的触发器时，可以删除该触发器，其指令如下所示。

```
DROP TRIGGER trigger_name
```

下面给出一个实例，删除触发器 backup_emp_trigger。

【实例 20-35】删除触发器 backup_emp_trigger。

```
SQL> drop trigger backup_emp_trigger;

触发器已删除。
```

为了验证是否成功删除触发器 backup_emp_trigger，再通过数据字典 USER_TRIGGERS 来查看该触发器的信息。

【实例 20-36】查看触发器 backup_emp_trigger 是否存在。

```
SQL> select trigger_name,triggering_event,status
  2  from user_triggers
  3  where trigger_name = 'BACKUP_EMP_TRIGGER';

未选定行
```

输出结果显示“未选定行”，说明数据字典中没有该触发器的记录，已成功删除触发器 backup_emp_trigger。

20.9 本章小结

本章介绍了数据库中非常重要的对象——触发器（TRIGGER），触发器允许 Oracle 基于业务规则或数据库安全考虑进行编码。触发器是使用 PL/SQL 语言编写的代码块，Oracle 允许使用触发器激发基于表和数据库的行为，如对表做 DML 操作时，使用触发器记录用户操作轨迹，当数据库启动或关闭时记录数据库的状态信息等。

本章在开始部分介绍了如何创建触发器，读者通过它可以对触发器建立感性认识。笔者基于操作类型对触发器进行了分类。在实际中审核触发器和删除触发器是具有现实意义的，略加修改读者可以创建符合自己业务需求的触发器。在触发器中允许使用条件语句进行触发器事件的细粒度控制，如 WHEN 子句和 IF 子句。而触发器管理是触发器维护的重要内容，本章的最后介绍了如何使用数据字典查看触发器的内容，以及重新编译触发器、屏蔽触发器和删除触发器。

第 21 章

序列号和同义词

序列号和同义词是 Oracle 中两个很有用的对象，序列号生成器会自动管理序列号，对于某些订单系统很有用处。使用同义词对象可以创建很多对象，如表、索引、函数、过程等的同义词，使用同义词使得操作的对象更加易于理解，同时使用同义词可以方便用户访问属于其他用户的数据库对象，或出于安全目的，因为同义词名字可以随意命名，没有限制，这样就隐藏了创建同义词的原始对象的信息。

21.1 序列号

Oracle 使用序列生成器自动产生用户可以在事务中使用的唯一序列号，该序列号是一个整数类型数据，序列生成器主要完成在多用户环境下产生唯一的数字序列，但是不会造成额外的磁盘 I/O 或事务锁。简单地说序列号是 Oracle 数据库的一个对象，该对象产生唯一序列号。

在不使用序列号时，如果多个用户同时向 EMP 表中插入一条员工记录，用户必须等待以得到下一个可用的员工号，而一旦使用序列号，则用户无须相互等待就可以得到下一个可用的员工号。序列生成器会自动为每个用户创建正确的员工编号。

可见使用序列生成器，避免了多用户相互等待而造成的事务串行执行，序列号的使用提高了系统的事务处理能力，减少多用户并行操作的等待时间。

Oracle 序列号具有如下特点：

- 序列号是独立于表的对象，由 Oracle 自动维护。
- 序列号可以对多个用户共享使用，即同一个序列对象供多个表使用且相互独立。
- 在 SQL 语句中使用序列号就可以使用它产生的序列号。

下面给出实例来说明如何创建和使用序列号以及序列号的维护工作。

21.1.1 创建和使用序列号

本节将讲解如何创建一个序列号，并给出使用序列号的实例来分析序列号的特点，如实例 21-1 所示。

【实例 21-1】创建序列号。

```
SQL> create sequence emp_seq
  2  start with 1000
  3  increment by 1
  21  nomaxvalue
  5  nocycle;

序列已创建。
```

提示"序列已创建"，此时使用数据字典 user_sequences 来查看刚刚创建的序列号的信息，如实例 21-2 所示。

【实例 21-2】使用数据字典 user_sequences 查看序列号信息。

```
SQL> col cycle_flag for a10
SQL> select sequence_name,min_value,increment_by,cycle_flag
  2  from user_sequences
  3* where sequence_name like 'EMP%'

SEQUENCE_NAME                 MIN_VALUE INCREMENT_BY CYCLE_FLAG
------------------------------------------------------ ----------------- -
EMP_SEQ                                1           1          N
```

实例 21-2 的结果说明刚刚创建的序列号 EMP_SEQ 已经记录在系统当中，接着就可以使用该序列号为我们服务了。

这里给出创建序列号的语句格式，使得读者可以全面地了解创建序列号的各种参数以及含义。创建序列号的语句格式如下：

```
CREATE SEQUENCE sequence_name
[START WITH n]
[INCREMENT BY n]
[{MAXVALUE n|NOMAXVALUE}]
[{MINVALUE n|NOMINVALUE}]
[{CACHE n|NOCACHE}]
[{CYCLE n|NOCYCLE}]
```

下面依次解释各参数的含义。

- START WITH n：序列号初始值，默认值为 1。
- INCREMENT BY n：序列号的步进幅度，默认步进幅度为 1。
- MAXVALUE n：定义序列号的最大值。
- NOMAXVALUE：不设置序列号的最大值，但是计算机对于数据的表达是有限的，实际上这个值当序列号是升序时为 10^{27}，序列号为降序时为-1。
- MINVALUE n：定义序列号的最小值。
- NOMINVALUE：不定义序列号的最小值，但是和参数 NOMAXVALUE 对应该值，对于升序的序列号最小值为 1，对于降序的序列号的最小值为-10^{26}。
- CACHE n：Oracle 服务器会预分配 n 个序列号并保存在内存中。
- NOCACHE：Oracle 服务器不会预分配序列号并保存在内存中。

- CYCLE n：定义序列号在达到最大值或最小值后，将继续产生序列号。
- NOCYCLE：定义序列号在达到最大值或最小值后，不再产生序列号。

在使用序列号前，先创建一个表 EMPLOYEES，如实例 21-3 所示。

【实例 21-3】创建一个表 EMPLOYEES。

```
SQL> create table employees
  2  (employee id   number(6)  not null,
  3   emp_name      varchar2(20),
  21  email          varchar2(25) not null,
  5   phone_number  varchar2(20),
  6   hiredate      date    not null);

表已创建。
```

现在已经成功创建了一个表，在使用序列号前还有一点说明，即两个伪列的使用，一个为 curval，该伪列提供序列的当前值；另一个为 nextval，该伪列提供下一个序列号的值。下面演示如何使用序列号自动生成员工号，如实例 21-4 所示，向表中插入两行数据。

【实例 21-4】向表 employees 中插入两行数据。

```
SQL>  insert into employees
  2   values
  3   (emp_seq.nextval,'tom','susu@yahoo.com',13988383756,sysdate);

已创建 1 行。
SQL> insert into employees
  2  values
  3  (emp_seq.nextval,'larry','larry@yahoo.com',139832182756,sysdate);

已创建 1 行。
```

此时，已向表 employees 中插入了两行数据，并且员工号是通过序列号产生的，下面查看员工信息，以认识序列号产生的员工号，如实例 21-5 所示。

【实例 21-5】查看员工信息来认识序列号产生的员工号。

```
SQL> select *
  2  from employees;

EMPLOYEE ID EMP NAME     EMAIL      PHONE NUMBER          HIREDATE
----------- -------------------- ------------------------- ------------
1000          tom           susu@yahoo.com       13988383756            06-8 月 -09
1001          larry        larry@yahoo.com      139832182756           06-8 月 -09
```

我们看到 EMPLOYEE_ID 依次为 1000 和 1001，因为序列号 EMP_SEQ 从 1000 开始，步进为 1，所以每使用一次 EMP_SEQ 序列号就增 1。上面在插入数据时我们使用序列号连续插入两行数据，所以出现的员工号是连续的。

注意，序列号是不会逆转的，如果用户添加了一条记录，而后删除该记录，此时序列号已经递增了，下次再使用序列号产生员工号时，会出现不连续现象。

【实例 21-6】先删除一条记录再插入一条记录。

```
SQL> delete from employees
  2  where employee_id=1001;

已删除 1 行。
SQL>  insert into employees
  2   values
  3   (emp_seq.nextval,'asaf','asaf@yahoo.com',13983872756,sysdate);

已创建 1 行。
```

此时，已经成功插入一条记录，而该记录使用了 EMP_SEQ 序列号产生员工号，查看这个员工号的值，如实例 21-7 所示。

【实例 21-7】再次查询 EMPLOYEES 表的全部数据。

```
SQL> select *
  2  from employees;

EMPLOYEE_ID EMP_NAME        EMAIL          PHONE_NUMBER          HIREDATE
----------- -------------------- ------------------------- -----------
 1000 tom         susu@yahoo.com          13988383756          06-8 月 -09
1002 asaf         asaf@yahoo.com          13983872756          06-8 月 -09
```

此时出现了不连续的员工号，主要是因为删除了一个员工记录，而序列号又只增不减。但是无论如何使用序列号保持了员工号的唯一性。

在任何时候，都可以使用 currval 伪列查询当前序列号的值，使用 nextval 查询序列号的下一个值，如实例 21-8 所示，此时使用续表 DUAL。

【实例 21-8】使用 currval 伪列查询当前序列号的值。

```
SQL>select emp_seq.currval,emp_seq.nextval
  2* from dual

   CURRVAL    NEXTVAL
---------------------- ------------------
      1002       1003
```

21.1.2　修改序列号

随着业务的变化，某些时候需要对已经定义的序列号进行修改，Oracle 允许使用 ALTER SEQUENCE 语句完成对序列号的修改。在给出修改示例前，先查看数据字典 USER_SEQUENCE 的属性信息，以了解修改序列号的那些参数。如实例 21-9 所示。

【实例 21-9】查看数据字典 USER_SEQUENCE 的属性信息。

```
SQL> desc user_sequences;
 名称                                              是否为空? 类型
 ----------------------------------------------------- --------
 SEQUENCE_NAME                                   NOT NULL VARCHAR2(30)
```

```
MIN_VALUE                                        NUMBER
MAX_VALUE                                        NUMBER
INCREMENT_BY                            NOT NULL NUMBER
CYCLE_FLAG                                       VARCHAR2(1)
ORDER_FLAG                                       VARCHAR2(1)
CACHE_SIZE                              NOT NULL NUMBER
LAST_NUMBER                             NOT NULL NUMBER
```

在修改参数 CACHE_SIZE 和 INCREMENT_BY 之前，先查询当前的序列号 EMP_SEQ 的相关属性信息。如实例 21-10 所示。

【实例 21-10】查询当前的序列号 EMP_SEQ 的相关属性信息。

```
SQL> select cache_size,increment_by,cycle_flag
  2  from user_sequences
  3* where sequence_name like 'EMP%'

CACHE_SIZE INCREMENT_BY CYCLE_FLAG
---------------------- -------------- --------------------------
      20           1        N
```

我们看到序列号 EMP_SEQ 的 CACHE_SIZE 为 20，而 INCREMENT_BY 的值为 1，现在修改序列号 EMP_SEQ 的参数设置，如实例 21-11 所示。

【实例 21-11】修改序列号 EMP_SEQ 的参数设置。

```
SQL> alter sequence emp_seq
  2  increment by 2
  3  cache 30;

序列已更改。
```

我们已经成功修改了序列号 EMP_SEQ 的参数设置，将参数 INCREMENT_BY 改为 2，而将 CACHE_SIZE 设置为 30。继续使用数据字典 USER_SEQUENCES 查看修改后的参数，如实例 21-12 所示。

【实例 21-12】使用数据字典 USER_SEQUENCES 查看修改后的参数。

```
SQL> select cache_size,increment_by,cycle_flag
  2  from user_sequences
  3  where sequence name like 'EMP%';

CACHE_SIZE INCREMENT_BY CYCLE_FLAG
---------------------- -------------- --------------------------
      30           2        N
```

此时，再向表 EMPLOYEES 中插入一行数据，则序列号步进为 2，因为当前的序列号值为 1002，所以 nextval 为 10021。插入一行数据，如实例 21-13 所示。

【实例 21-13】向表 employees 插入一行数据。

```
SQL> insert into employees
  2  values
```

```
  3  (emp_seq.nextval,'susu','asaf@yahoo.com',13983872756,sysdate);

已创建 1 行。
```

此时成功插入一行数据，再使用实例 21-14 查看插入记录的员工号。

【实例 21-14】查看插入记录的员工号。

```
SQL> select *
  2  from employees;

EMPLOYEE_ID EMP_NAME       EMAIL       PHONE_NUMBER       HIREDATE
----------- -------------------- ------------------------- ------
1000 tom                susu@yahoo.com        13988383756          06-8 月 -09
1002 asaf               asaf@yahoo.com        13983872756          06-8 月 -09
10021 susu               asaf@yahoo.com        13983872756          06-8 月 -09
```

显然此时的 EMPLOYEE_ID 为 10021，在当前值 1002 的基础上步进为 2。因为修改了序列号 EMP_SEQ 的参数设置，并且当前的序列号在内存中会放置 30 个值。

查看该序列号的下一个值，如实例 21-15 所示。

【实例 21-15】查看该序列号的值。

```
SQL> select emp_seq.nextval
  2  from dual;

   NEXTVAL
-------------------
     1006
```

显然这个值和步进参数（INCREMENT_BY）为 2 是一致的。

最后给出修改序列号的语句格式，如下所示：

```
ALTER SEQUENCE sequence name
[INCREMENT BY n]
[{MAXVALUE n|NOMAXVALUE}]
[{MINVALUE n|NOMINVALUE}]
[{CACHE n|NOCACHE}]
[{CYCLE n|NOCYCLE}]
```

下面解释一下各参数的含义。

- INCREMENT BY n：修改序列号每次执行的增长幅度。
- MAXVALUE n|NOMAXVALUE：修改序列号的最大值，不设置最大值上限。
- MINVALUE n|NOMINVALUE：修改序列号的最小值，不设置最小值下限。
- CACHE n|NOCACHE：修改序列号在内存预分配并保存在内存中的个数。
- CYCLE n|NOCYCLE：修改序列号使得序列号达到最大值或最小值后可以继续产生序列号。



修改序列号的用户必须拥有必要的权限，不能修改 START WITH 参数，ALTER SEQUENCE 对于之前产生的序列号没影响，只影响之后产生的序列号。

21.1.3 删除序列号

当不需要一个序列号时，可以使用 DROP SEQUENCE 指令删除该序列号，下面删除序列号 EMP_SEQ，如实例 21-16 所示。

【实例 21-16】删除序列号 EMP_SEQ。

```
SQL> drop sequence emp_seq;

序列已删除。
```

显示序列已删除，使用数据字典 USER_SEQUENCES 来验证是否成功删除序列号 EMP_SEQ，如实例 21-17 所示。

【实例 21-17】验证是否成功删除序列号 EMP_SEQ。

```
SQL> select sequence_name,increment_by,cache_size
  2  from user_sequences
  3  where sequence_name = 'EMP_SEQ';

未选定行
```

显然数据字典中没有符合条件的记录，说明已经成功删除序列号 EMP_SEQ，下面如果再向表 EMPLOYEES 中插入一行记录，并使用序列号，则会产生错误，如实例 21-18 所示。

【实例 21-18】向表 EMPLOYEES 中插入一行记录验证是否删除序列号。

```
SQL> insert into employees
  2  values
  3  (emp_seq.nextval,'susu','asaf@yahoo.com',13983872756,sysdate);
(emp_seq.nextval,'susu','asaf@yahoo.com',13983872756,sysdate)
 *
第 3 行出现错误:
ORA-02289: 序列不存在
```

因为序列对象 EMP_SEQ 已经删除，所以无法使用序列对象，从而无法实现使用序列号的插入操作。

21.2 同义词

同义词是 Oracle 数据库中对象的别名，在 Oracle 数据库中对象包括表、视图、物化视图（Oracle10g）、触发器、序列号、函数、过程以及 Java 对象等，Oracle 可以为这些对象创建同义词。

使用同义词的主要目的是方便用户访问属于其他用户的数据库对象，或出于安全目的，因为同义词名字具有随机性，没有限制，这样就隐藏了创建同义词的原始对象的信息。同义词可以是公有的，也可以是私有的。顾名思义，公有同义词是任何用户都可以使用的，而私有同义词只有指定的用户可以使用，没有授权的其他用户无法访问某一个用户的私有同义词。

下面通过实例分析同义词的优点，如实例 21-19 所示。

【实例 21-19】在 SYSTEM 用户下查看 SCOTT 用户的 DEPT 表信息。

```
SQL> conn system/oracle
已连接。
SQL> select *
  2  from dept;
from dept
     *
第 2 行出现错误:
ORA-009212: 表或视图不存在
```

显然在 SYSTEM 用户模式下，无法识别 SCOTT 用户的 DEPT，而必须在表 DEPT 前使用 SCOTT 用户模式，说明该表属于 SCOTT 用户，如实例 21-20 所示。

【实例 21-20】通过指定表 DEPT 的模式名 SCOTT 查看表信息。

```
SQL> conn system/oracle
已连接。
SQL> select *
  2  from scott.dept;

   DEPTNO DNAME          LOC
------------------- ---------------------- --------------
       210 OPERATIONS     BOSTON
       10 ACCOUNTING    NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
```

可见，使用模式名确实实现了查询，但是却对使用者"暴露"了表 DEPT 的模式信息。而使用别名则可以避免这个问题。下面介绍如何创建同义词。

21.2.1　创建公有同义词

公有同义词对数据库的所有用户有效，一般是某种应用的所有者创建如过程或程序包的公有同义词，以便其他用户可以使用它们。下面先给出实例说明如何创建公有同义词，如实例 21-21 所示。

【实例 21-21】创建公有同义词。

```
SQL> create public synonym department for scott.dept;

同义词已创建。
```

在上例成功创建同义词后，下面就可以使用公有同义词 department 了，如实例 21-22 所示。

【实例 21-22】使用公有同义词查询数据。

```
SQL> conn system/oracle
已连接。
SQL> select *
  2  from department;

   DEPTNO DNAME          LOC
```

```
---------- -------------- -------------
       210 OPERATIONS     BOSTON
       10  ACCOUNTING    NEW YORK
       20  RESEARCH       DALLAS
       30  SALES          CHICAGO
```

可见使用公有同义词避免了实例 21-19 所示的问题，此时任何用户使用公有同义词 department 都可以访问 SCOTT 用户的 DEPT 表了。

为了更有说服力，我们使用 SCOTT 用户登录，看是否可以使用该同义词，如实例 21-23 所示。

【实例 21-23】使用 SCOTT 用户验证使用公有同义词。

```
SQL> conn scott/oracle
已连接。
SQL> select *
  2  from department;

   DEPTNO DNAME          LOC
---------- -------------- -------------
       210 OPERATIONS     BOSTON
       10  ACCOUNTING    NEW YORK
       20  RESEARCH       DALLAS
       30  SALES          CHICAGO
```

现在使用公有同义词，在 SCOTT 用户模式下获得需要的数据，其实读者也可以通过其他用户自行验证。



创建公有同义词的用户必须具有 CREATE PUBLIC SYNONYM 的权限，当其他用户使用该公有同义词时，只有具有相应的权限才可以对同义词对应的表执行诸如 DELETE、UPDATE 等操作。

21.2.2 创建私有同义词

私有同义词和公有同义词相对，私有同义词只对创建它的用户有效，而其他用户无法使用，如实例 21-24 所示。

【实例 21-24】创建私有同义词。

```
SQL> create synonym d for scott.dept;

同义词已创建。
```

此时，已成功创建一个私有同义词 d，该同义词代表 SCOTT 用户的对象，即表 DEPT。下面在 SYSTEM 用户模式下，使用同义词 d，如实例 21-25 所示。

【实例 21-25】验证使用私有同义词。

```
SQL> conn system/oracle
已连接。
SQL> select *
```

```
  2  from d;

   DEPTNO DNAME          LOC
----------------- ------------------------- -------------
       210 OPERATIONS     BOSTON
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
```

再使用 SCOTT 用户登录数据库，也试图使用私有同义词 d，如实例 21-26 所示。

【实例 21-26】在 SCOTT 用户模式下验证私有同义词 d。

```
SQL> conn scott/oracle
已连接。
SQL> select *
  2  from d;
from d
     *
第 2 行出现错误:
ORA-009212: 表或视图不存在
```

我们看到私有同义词 d 在 SCOTT 用户模式下无法使用。



创建私有同义词和创建公有同义词的唯一区别就是是否使用 PUBLIC 关键字，如果不使用 PUBLIC 关键字说明是私有同义词，如果使用 PUBLIC 关键字说明是公有同义词。

21.2.3　删除同义词

删除同义词的指令很简单，即使用 DROP 关键字，但是删除公有同义词时必须使用 PUBLIC 关键字，而删除私有同义词时就不需要。在 SYSTEM 用户下创建了一个公有同义词 deparment 和一个私有同义词 d，下面删除这两个同义词，分别如实例 21-27 和实例 21-28 所示。

【实例 21-27】删除公有同义词 department。

```
SQL> drop public synonym department;

同义词已删除。
```

【实例 21-28】删除私有同义词 d。

```
SQL> conn system/oracle
已连接。
SQL> drop synonym d;

同义词已删除。
```

21.3　本章小结

本章介绍了两个很有用的 Oracle 对象——序列号和同义词，通过本章的学习读者可以把握 Oracle 引入序列号的初衷，以及如何创建和使用序列号。序列号的维护主要包括删除和修改序列号。而同义词确实方便了用户对其他用户的对象的使用，使用公有同义词使得任何用户都可以操作某个数据库对象，如果出于安全考虑，则可使用私有同义词或者使用授权的方式，将用户创建的同义词授权给信任的用户。

第 22 章

用户管理和概要文件

Oracle 通过设置用户来访问数据库，对用户赋予不同的资源使得用户具有操作数据库的不同权限。在数据库被使用前，必须由数据库管理员创建用户和用户登录密码，随后可以由 DBA 或其他具有 DBA 权限的用户授予新创建的用户访问各种资源的权限。

22.1 用户管理

22.1.1 创建用户

在 Oracle 中必须使用用户登录数据库，通过设置密码完成用户身份认证，一旦登录数据库，该用户就可以访问它拥有的数据库对象，最明显的实例是访问其中的表对象。下面先给出一个实例说明如何创建用户，使得读者对用户创建有直观认识，然后再通过创建用户的语法详细介绍一些参数设置。

1. 创建新用户

要创建数据库必须使用 DBA 权限的用户，本例中使用 SYSTEM 用户登录数据库，密码为 ORACLE，读者的密码或许有所不同，在安装 Oracle 10g 数据库过程中会提示解锁的用户和设置密码，在 Oracle 9i 中 SYSTEM 用户密码默认为 MANAGER。

【实例 22-1】使用 SYSTEM 用户登录数据库。

```
SQL> connect system/oracle@orcl
已连接。
```

【实例 22-2】创建数据库用户。

```
SQL> create user jane
  2  identified by american
  3  default tablespace users
  4  temporary tablespace temp
  5  quota 10m on users
```

```
  6  password expire;

用户已创建。
```

在上例中，创建了用户 jane，下面具体分析每行的含义。

- create user jane：创建用户 jane，其中 create user 为创建用户指令。
- identified by american：设置用户密码为 american，其中 identified by 为创建用户指令。
- default tablespace users：设置默认表空间为 USERS 表空间，该表空间存储用户数据。
- temporary tablespace temp：创建临时表空间 TEMP，该表空间用于诸如排序等操作的数据空间。
- quota 10m on users：设置该用户对于表空间 USERS 的配额为 10M。
- password expire：说明用户 jane 登录数据库时，密码立即失效，Oracle 会提示重新输入密码，如下所示：

```
SQL> connect jane/american@orcl
ERROR:
ORA-28001: the password has expired
更改 jane 的口令
新口令:
重新键入新口令:
ERROR:
ORA-01045: user JANE lacks CREATE SESSION privilege; logon denied

口令未更改
警告: 您不再连接到 ORACLE。
```

此时，由于在创建用户 jane 时，没有赋予该用户 CREATE SESSION 权限，所以虽然更改了用户密码，还是无法建立与数据库的会话连接。需要向用户授权，否则该用户就是“花瓶”，在赋予该用户权限后，该用户就可以使用新的用户密码登录数据库了，如实例 22-3 所示。

【实例 22-3】为用户 jane 赋予 CREATE SESSION 权限。

```
SQL> connect system/oracle@orcl
已连接。
SQL> grant create session,resource to jane;

授权成功。
```

然后就可以用新密码登录数据库了，如下所示。

```
SQL> connect jane/jane@orcl
已连接。
```

在成功创建了数据库后如何查看创建的用户诸如表空间等信息呢？答案是使用数据字典 DBA_USERS。此时，需要以 DBA 角色的用户登录数据库，如实例 22-4 所示。

【实例 22-4】使用数据字典 DBA_USERS 查看用户 jane 的信息。

```
SQL> col username for a10
SQL> col default_tablespace for a10
```

```
SQL> col temporary_tablespace for a15
SQL> col password for a20
SQL>                                                              select
username,password,expiry_date,default_tablespace,temporary_tablespace,created
  2  from dba_users
  3* where username = 'JANE'

USERNAME   PASSWORD   EXPIRY_DATE   DEFAULT_TA TEMPORARY_TABLE CREATED
---------- -------------------- ------------------------------------ ---
JANE      4968BF82C7B085C6                 USERS      TEMP            03-9 月 -09
```

从实例 22-4 的输出可以看出用户的默认表空间为 USERS，而临时表空间为 TEMP，该用户的创建时间为 03-9 月-09。而密码是加密的，这也是 Oracle 认为安全第一的缘故，即使具有 DBA 权限的用户也无法看到该用户的密码，虽然 DBA 用户可以创建或删除用户。

在创建用户时，使用了 QUOTA 参数，设置该用户只能使用表空间 USERS 的 10M 空间，使用数据字典 DBA_TS_QUOTAS 可以查看用户 JANE 在表空间上的配额信息。

【实例 22-5】查看用户 jane 的表空间配额信息。

```
SQL> select tablespace_name,username, max_bytes
  2  from dba_ts_quotas
  3  where username ='JANE';

TABLESPACE_NAME                USERNAME     MAX_BYTES
------------------------------------------------------ ----------------------
USERS                           JANE          10485760
```

2. 创建用户语法及参数含义

前面通过实例体验了如何创建一个新用户，其实读者只需要修改个别参数，如表空间的名字（可能需要用户事先创建表空间）就可以直接应用来创建自己的用户。下面是创建用户的详细语法，如下所示。

【实例 22-6】创建用户的语法格式。

```
CREATE USER user
IDENTIFIED {BY password | EXTERNALLY}
[DEFAULT TABLESPACE tablespace]
[TEMPORARY TABLESPACE tablespace]
[QUOTA {integer [K | M]| UNLIMITED} ON tablespace
[QUOTA {integer [K | M]| UNLIMITED} ON tablespace
]……]
[PASSWORD EXPIRE]
[ACCOUNT { LOCK | UNLOCK }]
[PROFILE { profile | DEFAULT }]
```

下面介绍其中的几个参数。

- CREATE USER user：创建用户 user。
- IDENTIFIED{BY password | EXTERNALLY}：设置用户密码，EXTERNALLY 说明该用户由操作系统授权。该参数在创建用户时是不能省略的。

- DEFAULT TABLESPACE tablespace：设置用户的默认表空间。
- TEMPORARY TABLESPACE tablespace：设置用户的临时表空间。
- QUOTA {integer[K|M]|UNLIMITED} ON tablespace：设置该用户对于表空间的配额，即表空间的多大空间给该用户使用，参数 UNLIMITED 说明没有限制，K|M 是配额单位。
- PASSWORD EXPIRE: 设置用户密码在用户第一次使用时作废，需要重新设置该用户密码。
- ACCOUNT{LOCK | UNLOCK}：选择是否锁定该用户，LOCK 为锁定用户，而 UNLOCK 为不锁定用户，该参数的默认值是 UNLOCK。
- PROFILE {profile |DEFAULT}：使用指定的概要文件，profile 为概要文件名。如果不指定概要文件，则使用 DEFAULT 的默认概要文件，默认的概要文件对所有限制的初始值没有限制。

从以上创建用户的语法可以看出，在创建新用户前，必须做些准备工作，整个准备工作和创建过程如下所示。

- 确认存储用户对象的表空间。
- 确定在每个表空间上的空间配额。
- 分配一个默认表空间和一个临时表空间。
- 开始创建用户。
- 向用户授权和角色，如使得用户具有建立会话的权利，辅以用户 DBA 角色权限等。

3. 改变用户参数

在成功创建用户后，如果对用户参数如默认表空间等不满意，可以改变用户参数，如修改用户 jane 的默认表空间或者修改当前默认表空间的配额。

【实例 22-7】修改用户 jane 的默认表空间配额。

```
SQL> connect system/oracle@orcl
已连接。
SQL> alter user jane
  2  quota 20m on users;

用户已更改。
```

在修改成功后，可使用数据字典 DBA_TS_QUOTAS 来验证修改结果。

【实例 22-8】验证用户 jane 的表空间修改结果。

```
SQL> select tablespace_name,username,max_bytes
  2  from dba_ts_quotas
  3  where username ='JANE';

TABLESPACE_NAME       USERNAME                  MAX_BYTES
------------------------------------- ---------------
USERS                   JANE                      20971520
```

从输出结果可以看出用户 JANE 在表空间 USERS 上的配额被修改为 20M。说明实例 22-7 已修改成功。

在生产数据库中，会出现用户默认表空间不足的情况，在用户创建数据库对象如表、索引时，如果没有指定存储表空间就存放在创建用户时的默认表空间中，数据表空间的不足会造成数据库挂起，所以需要修改用户在默认临时表空间的 QUOTA 参数，而如何增加一个默认表空间呢？下面给出一个示例（用户事先创建表空间 NEWTBS）。

【实例 22-9】修改用户 jane 的默认表空间。

```
SQL> alter user jane
  2  default tablespace newtbs
  3  quota unlimited on system;

用户已更改。
```

上例中，增加了用户 jane 的一个默认表空间为 newtbs，在该表空间的配额为 unlimited（没有限制）。再通过数据字典 DBA_TS_QUOTAS 来查看修改结果。

【实例 22-10】查看用户 JANE 的默认表空间修改信息。

```
SQL> select username,tablespace_name,max_bytes
  2  from dba ts quotas
  3  where username ='JANE';

USERNAME                 TABLESPACE_NAME              MAX_BYTES
-------------------------------------- --------------------
JANE                       NEWTBS                              -1
JANE                       USERS                         20971520
```

从上述输出可以看出用户 JANE 在表空间 NEWTBS 上的配额为-1，说明没有限制，而此时用户在表空间 USERS 上的配额依然存在。如果不希望用户使用表空间 USERS 的空间，即回收用户在 USERS 表空间的使用权，又如何处理呢？如实例 22-11 所示。

【实例 22-11】回收用户 JANE 在表空间 USERS 的使用权。

```
SQL> alter user jane
  2  quota 0 on users;

用户已更改。
```

此时，通过设置用户在表空间 USERS 上的配额为 0 来回收对其使用权。然后再通过数据字典 DBA_TS_QUOTAS 来查看该用户的表空间配额信息，如下所示。

【实例 22-12】验证是否回收用户 JANE 的表空间 USERS 的使用权。

```
SQL>  select username,tablespace_name,max_bytes
  2   from dba_ts_quotas
  3   where username ='JANE';

USERNAME                 TABLESPACE_NAME             MAX_BYTES
-------------------------------------- ----------------
JANE                      NEWTBS                              -1
```

从输出可以看出用户 JANE 没有使用表空间 USERS，只有表空间 NEWTBS，而且该表空间的

使用空间不受限制（当然不能超过表空间中所有数据文件大小的总和，即使数据文件的大小可以自动扩展也不能超过磁盘空间的限制），如果在回收 USERS 表空间的使用权之前，已经在该表空间上使用了 5M 空间，则不能再给用户 JANE 分配空间使用了。

22.1.2 删除用户

删除用户的语法格式如下所示：

```
DROP USER user_name [CASCADE]
```

如果使用 CASCADE 参数说明要删除和用户相关的所有数据库对象，如触发器、外键索引、过程等。删除用户 JANE 如实例 22-13 所示。

【实例 22-13】删除用户 JANE。

```
SQL> drop user jane;

用户已删除。
```

下面，验证是否成功删除用户 JANE，如实例 22-14 所示。

【实例 22-14】验证用户 JANE 是否存在。

```
SQL> select username,created,default_tablespace
  2  from dba_users
  3  where username ='JANE';

未选定行
```

输出结果是“未选定行”，说明用户 JANE 不存在，在删除用户时，如果该用户已经连接到数据库服务器，则无法删除。可以断开该用户的连接后再删除用户。也可以使用数据字典 DBA_USERS 来查看当前系统上的用户名，如实例 22-15 所示。

【实例 22-15】查看当前系统上的所有用户信息。

```
SQL> select username,account status,created
  2  from dba_users;

USERNAME                     ACCOUNT_STATUS                 CREATED
------------------------------ ----------------------
LINZI                        OPEN                           28-8 月 -09
LAURENCE                     OPEN                           04-9 月 -09
SCOTT                        OPEN                           25-8 月 -09
CAT                          OPEN                           22-8 月 -09
HR                           OPEN                           12-7 月 -09
RMAN_BACKUP                  OPEN                           03-9 月 -09
TSMSYS                       EXPIRED & LOCKED               30-8 月 -05
BI                           EXPIRED & LOCKED               12-7 月 -09
PM                           EXPIRED & LOCKED               12-7 月 -09
............
OUTLN                        EXPIRED & LOCKED               30-8 月 -05
```

```
已选择 31 行。
```

在以上输出中 ACCOUNT_STATUS 说明用户的状态，其中值 OPEN 说明该用户可用，而 EXPIRED 说明该用户过期，LOCKED 说明该用户锁定，那么如何解锁这些锁定的用户呢？如实例 22-16 所示。

【实例 22-16】解锁用户。

```
SQL> ALTER USER outln IDENTIFIED BY outln ACCOUNT UNLOCK;

用户已更改。
```

在解锁过程中，需要先使用 IDENTIFIED BY 修改用户的密码，这也是 Oracle 安全理念的体现，无论何时安全第一。解锁了的用户就可以正常登录数据库了，如实例 22-17 所示。

【实例 22-17】使用解锁的用户登录数据库。

```
SQL> connect outln/outln@orcl
已连接。
SQL> show user
USER 为 "OUTLN"
```

22.2　概要文件

在创建用户后就需要给予该用户各种系统资源，如 CPU、并行会话数、空闲时间限制等资源限制，同时需要对口令做出更详细的管理方案，如尝试登录指定的次数后账户被锁定、口令过期之后的处理等，如果对每个用户都进行资源限制或口令管理，要输入大量的指令，如每个用户输入 10 个资源限制或口令限制指令，对 10 个用户就输入 100 条指令，显然这样的效率很低，尤其是对用户的资源限制和口令限制都相同时，只是重复地输入指令，Oracle 提供了概要文件来管理用户，可以避免上述问题。

22.2.1　什么是概要文件

概要文件就是一组指令的集合，这些指令限制了用户资源的使用或口令的管理，在创建用户时，有一个 PROFILE 参数就是用来指定概要文件的，一旦概要文件创建就可以将概要文件通过 ALTER USER 指令赋予用户或者在 CREATE USER 时指定概要文件。

通过将概要文件赋予用户可以极大较少 DBA 的工作量。如果没有指定概要文件，则会自动使用一个默认概要文件。

使用概要文件可以实现用户的资源管理和口令管理。操作步骤如下所示。

01 使用 CREATE PROFILE 指令创建一个概要文件。

02 使用 ALTER USER（已有用户）或 CREATE USER（新用户）将概要文件赋予用户。

03 （对于资源管理而言）启动资源限制，修改动态参数 RESOURCE_LIMIT 为 TRUE，此时既可以通过修改参数文件，也可以使用 ALTER SYSTEM 来修改。

22.2.2 使用概要文件管理会话资源

当用户连接到数据库时，就与数据库服务器建立了会话连接，此时用户会消耗数据库服务器的资源，所以可创建一个会话级的数据库资源限制的概要文件来限制用户对资源的使用。先给出创建资源管理的概要文件的语法格式，如下所示：

```
CREATE PROFILE  profile_name LIMIT
[SESSIONS_PER_USER n]
[CPU PER SESSION n]
[CPU_PER_CALL n]
[CONNECT_TIME n]
[IDLE_TIME n]
[LOGICAL READS PER SESSION n]
[LOGICAL_READS_PER_CALL n]
```

其中 n 为最大值。下面逐行介绍每个资源限制的含义。

- SESSIONS_PER_USER n：表示每个用户的最大会话数。
- CPU_PER_SESSION：每个会话占用的 CPU 时间，单位是 0.01 秒。
- CPU_PER_CALL n：每个调用占用的 CPU 时间，单位是 0.01 秒。
- CONNECT_TIME n：每个支持连接的时间。
- IDLE_TIME n：每个会话的空闲时间。
- LOGICAL_READS_PER_SESSION n：每个会话的物理和逻辑读数据块数。

下面创建一个资源限制文件，如实例 22-18 所示。

【实例 22-18】创建资源限制概要文件。

```
SQL> create profile scott_prof limit
  2  sessions_per_user 10
  3  cpu per session 10000
  4  idle_time 40
  5  connect_time 120;

配置文件已创建
```

该资源限制文件创建了一个名为 SCOTT_PROF 的概要文件，加在该文件上的限制是 sessions_per_user，每个用户的并行会话数为 10，cpu_per_session 每个会话的 CPU 时间为 100 秒。idle_time 连接空闲时间为 40 分，connect_time 保持连接时间为 120 分。

通过数据字典 DBA_PROFILES 来查看刚刚创建的概要文件 SCOTT_PROF，如实例 22-19 所示。

【实例 22-19】查看概要文件 SCOTT_PROF。

```
SQL> col profile for a20
SQL> col resource_name for a25
SQL> col limit for a20
SQL> select *
  2  from dba_profiles
  3  where profile ='SCOTT_PROF'
```

```
  4  order by limit;

PROFILE              RESOURCE NAME            RESOURCE   LIMIT
-------------------------------- -------------------------------------------- --------
SCOTT_PROF           SESSIONS_PER_USER        KERNEL       10
SCOTT PROF           CPU PER SESSION          KERNEL       10000
SCOTT_PROF           CONNECT_TIME             KERNEL       120
SCOTT_PROF           IDLE_TIME                KERNEL        40
SCOTT_PROF           LOGICAL_READS_PER_SESSION KERNEL      DEFAULT
SCOTT PROF           LOGICAL READS PER CALL    KERNEL      DEFAULT
SCOTT_PROF           PASSWORD_GRACE_TIME       PASSWORD   DEFAULT
SCOTT_PROF           PRIVATE_SGA               KERNEL      DEFAULT
SCOTT PROF           COMPOSITE LIMIT           KERNEL      DEFAULT
SCOTT_PROF           PASSWORD_LIFE_TIME        PASSWORD   DEFAULT
SCOTT_PROF           PASSWORD_REUSE_TIME       PASSWORD   DEFAULT

PROFILE              RESOURCE NAME            RESOURCE LIMIT
-------------------------------- -------------------------------------------- -------
SCOTT_PROF           PASSWORD_REUSE_MAX        PASSWORD  DEFAULT
SCOTT_PROF           PASSWORD_VERIFY_FUNCTION  PASSWORD  DEFAULT
SCOTT PROF           PASSWORD LOCK TIME        PASSWORD  DEFAULT
SCOTT_PROF           CPU_PER_CALL              KERNEL     DEFAULT
SCOTT_PROF           FAILED_LOGIN_ATTEMPTS     PASSWORD  DEFAULT

已选择 16 行。
```

从输出可以看出概要文件 SCOTT_PROF 的所有资源参数，其中资源参数 SESSION_PER_USER、CPU_PER_SESSION、CONNECT_TIME、IDLE_TIME 为创建概要文件时指定的值，而其他资源参数都采用默认值。其中 RESOURCE 列的值中 KERNEL 行表示一个资源参数，而 PASSWORD 表示一个安全限制，接下来介绍如何创建口令管理的概要文件。

22.2.3　创建口令管理的概要文件

1. 口令管理参数以及含义

创建口令管理的概要文件与创建资源限制的概要文件一样，都是使用 CREATE USER 或者 ALTER USER 指令将概要文件赋予用户，口令文件一旦赋予用户立即生效，不需要开启设置。下面介绍完成口令管理的参数以及含义。先查看概要文件 SCOTT_PROF 中的口令参数，重新使用数据字典 DBA_PROFILES 查看概要文件 SCOTT_PROF 如实例 22-20 所示。

【实例 22-20】查看 SCOTT_PROF 概要文件的口令管理参数。

```
SQL> col resource_name for a30
SQL> select *
  2  from dba_profiles
  3  where profile='SCOTT_PROF'
  4* and resource_type='PASSWORD'

PROFILE         RESOURCE_NAME      RESOURCE_TYPE            LIMIT
```

```
-------------------- ------------------------------ -------
SCOTT_PROF   FAILED_LOGIN_ATTEMPTS      PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_LIFE_TIME         PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_REUSE_TIME        PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_REUSE_MAX         PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_VERIFY_FUNCTION   PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_LOCK_TIME         PASSWORD      DEFAULT
SCOTT_PROF   PASSWORD_GRACE_TIME        PASSWORD      DEFAULT

已选择 7 行。
```

从输出可以知道有 7 个参数用于实现用户的口令管理，如下所示。

- FAILED_LOGIN_ATTEMPTS：尝试失败登录的次数，如果用户登录数据库时登录失败次数超过该参数的值，则锁定该用户。
- PASSWORD_LIFE_TIME：口令有效的时限，超过该参数指定的天数，则口令失效。
- PASSWORD_REUSE_TIME：口令在能够重用之前的天数。
- PASSWORD_REUSE_MAX：口令能够重用之前的最大变化数。
- PASSWORD_LOCK_TIME：当用户登录失败后，用户被锁定的天数。
- PASSWORD_GRACE_TIME：口令过期之后还可以继续使用的天数。
- PASSWORD_VERIFY_FUNCTION：在为一个新用户赋予口令之前要验证口令的复杂性是否满足要求的一个函数，该函数使用 PL/SQL 语言编写，名字为 verify_function，该函数将做如下检查：①口令的最小长度要求 4 个字符；②口令不能与用户名相同；③口令应至少包含一个字符、一个数字和一个特殊字符。数字包括'0123456789'，字符包括'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'，而特殊字符包括'!"#$%&()``*+,-/:;<=>?_'；④新口令至少有 3 个字母与旧口令不同。

要使用 Oracle 提供的口令验证函数，需要先运行一个名为 utlpwdmg.sql 的脚本文件，执行脚本文件创建口令复杂性验证函数时，需要使用 SYS 用户登录数据库且作为 DBA 用户，该文件存储在 F:\oracle\product\10.2.0\db_1\RDBMS\ADMIN 目录下（根据安装的磁盘略有不同）。

【实例 22-21】执行创建口令复杂性验证函数的过程。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL> @F:\oracle\product\10.2.0\db_1\RDBMS\ADMIN\utlpwdmg.sql

函数已创建。


配置文件已更改
```

从输出可以看出，函数已经创建，且配置文件已经更改，这里创建了函数 verify_function。查看该函数是否存在，如实例 22-22 所示。

【实例 22-22】验证函数 verify_function 是否创建。

```
SQL> col owner for a10
```

```
SQL> col object_name for a20
SQL> select owner ,object_name,object_type,created
  2  from dba_objects
  3  where object_type ='FUNCTION'
  4* and object_name ='VERIFY_FUNCTION'

OWNER      OBJECT_NAME          OBJECT_TYPE        CREATED
------------------- ------------------------------------ -----------
SYS        VERIFY_FUNCTION      FUNCTION           05-9月 -09
```

显然，函数 VERIFY_FUNCTION 存在说明创建成功，那么“配置文件已经更改”是什么意思呢？也就是说在实例 22-22 中不但创建了函数，而且还更改了默认的概要文件，此时查看脚本文件 utlpwdmg.sql 就一目了然了，如下代码是脚本中的最后部分：

```
-- This script alters the default parameters for Password Management
-- This means that all the users on the system have Password Management
-- enabled and set to the following values unless another profile is
-- created with parameter values set to different value or UNLIMITED
-- is created and assigned to the user.

ALTER PROFILE DEFAULT LIMIT
PASSWORD_LIFE_TIME 60
PASSWORD_GRACE_TIME 10
PASSWORD_REUSE_TIME 1800
PASSWORD_REUSE_MAX UNLIMITED
FAILED_LOGIN_ATTEMPTS 3
PASSWORD_LOCK_TIME 1/1440
PASSWORD_VERIFY_FUNCTION verify_function;
```

这部分脚本代码改变了口令管理的默认概要文件，这意味着整个数据库系统的用户都使用在 ALTER PROFILE DEFAULT LIMIT 中设置的口令限制，除非用户创建了另一个口令管理的概要文件，或修改了概要文件参数值。

此时更改用户 SCOTT 的用户密码为 oracle，并查看是否已成功修改。

【实例 22-23】修改用户 SCOTT 的用户密码为 oracle。

```
SQL> conn system/oracle as sysdba
已连接。
SQL> alter user scott
  2  identified by oracle
  3  ;
alter user scott
*
第 1 行出现错误:
ORA-28003: 指定口令的口令验证失败
ORA-20002: Password too simple
```

显然，修改失败，因为密码 oracle 不符合函数 verify_function 中定义的规则之一。此时也说明口令管理的概要文件已即时生效。

使用 Oracle 的口令函数一般可以满足用户要求，如果用户需要也可以自己创建口令函数，这

里不再详细介绍，读者可以参考其他书籍。

2. 创建口令管理的概要文件的方法

在介绍了口令管理的参数以及含义后，就可以根据业务需要创建口令管理概要文件。如同创建资源限制的概要文件一样，语法格式如下所示。

```
CREATE PROFILE profile_name LIMIT
[parameter1 para value1]
[parameter2 para_value2]
……
```

【实例 22-24】创建口令管理的概要文件。

```
SQL> create profile password_prof limit
  2  failed_login_attempts 5
  3  password_life_time 90
  4  password reuse time 30
  5  password_lock_time 15
  6  password_grace_time 3;

配置文件已创建
```

上面创建了概要文件 password_prof，各个参数的含义如下所示。

- failed_login_attempts 5：尝试登录的失败次数为 5，5 次之后该用户将被锁定。
- password_life_time 90：该密码在 90 天内有效。
- password_reuse_time 30：该口令失效后 30 天后才可以使用。
- password_lock_time 15：在尝试登录指定的次数后，该用户被锁定 15 天。
- password_grace_time 3：在口令过期后，4 天内可以使用旧口令（过期的口令）登录数据库。

现在读者已经明白了实例 22-23 中创建的密码概要文件的作用了，下面通过数据字典 DBA_PROFILES 查看密码概要文件 PASSWORD_PROF 的口令参数设置。

【实例 22-25】查看密码概要文件 PASSWORD_PROF 的口令参数。

```
SQL> col resource_type for a10
SQL> col limit for a15
SQL>select *
  2   from dba_profiles
  3   where profile ='PASSWORD_PROF'
  4*  and resource_type ='PASSWORD'

PROFILE              RESOURCE_NAME              RESOURCE_T LIMIT
-------------------------------- -----------------------------------
PASSWORD_PROF     FAILED_LOGIN_ATTEMPTS     PASSWORD    5
PASSWORD PROF     PASSWORD LIFE TIME         PASSWORD    90
PASSWORD_PROF     PASSWORD_REUSE_TIME       PASSWORD    30
PASSWORD_PROF     PASSWORD_REUSE_MAX       PASSWORD   DEFAULT
PASSWORD PROF     PASSWORD VERIFY FUNCTION PASSWORD    DEFAULT
PASSWORD_PROF     PASSWORD_LOCK_TIME        PASSWORD    15
```

```
PASSWORD_PROF     PASSWORD_GRACE_TIME       PASSWORD   3

已选择 7 行。
```

从实例 22-25 的输出可以看出在创建密码概要文件时，没有明确给出的数值都采用默认值，这些参数 LIMIT 列的值为 DEFAULT。

22.2.4　修改和删除概要文件

Oracle 允许使用 ALTER PROFILE 指令来修改概要文件中的参数，如概要文件 PASSWORD_PROF 的口令管理参数。

【实例 22-26】修改口令管理概要文件的参数。

```
SQL> alter profile password_prof limit
  2  failed_login_attempts 3
  3  password_life_time 60
  4  password grace time 7;

配置文件已更改
```

输出显示已成功修改口令管理的配置文件，下面使用数据字典 DBA_PROFILES 来验证修改结果。

【实例 22-27】查看修改后的口令管理概要文件 PASSWORD_PROF 的参数。

```
SQL>  select *
  2   from dba_profiles
  3   where profile ='PASSWORD_PROF'
  4  and resource_type ='PASSWORD';

PROFILE            RESOURCE_NAME             RESOURCE_T LIMIT
------------------------------- ----------------------------------------
PASSWORD_PROF     FAILED_LOGIN_ATTEMPTS   PASSWORD   3
PASSWORD_PROF     PASSWORD_LIFE_TIME       PASSWORD   60
PASSWORD_PROF     PASSWORD_REUSE_TIME       PASSWORD   30
PASSWORD_PROF     PASSWORD_REUSE_MAX       PASSWORD  DEFAULT
PASSWORD_PROF     PASSWORD VERIFY FUNCTION PASSWORD  DEFAULT
PASSWORD_PROF     PASSWORD_LOCK_TIME         PASSWORD   15
PASSWORD_PROF     PASSWORD_GRACE_TIME      PASSWORD   7

已选择 7 行。
```

上述输出中字号加粗的行参数被修改成功，此时只是为了读者方便观察而设置的字体加粗。

如果不需要某个概要文件，可以使用指令 DROP PROFILE 删除，如果要删除的概要文件已经赋予了用户，则需要使用 CASCADE 参数。

【实例 22-28】删除概要文件 PASSWORD_PROF。

```
SQL> drop profile password_prof;

配置文件已删除。
```

验证是否成功删除口令管理的概要文件 PASSWORD_PROF，如实例 22-29 所示。

【实例 22-29】验证是否删除概要文件 PASSWORD_PROF。

```
SQL> select *
  2  from dba_profiles
  3  where profile ='PASSWORD_PROF';

未选定行
```

“未选定行”说明成功删除了概要文件 PASSWORD_PROF，因为在数据字典 DBA_PROFILES 中没有该文件记录。

22.3　本章小结

用户管理是 Oracle 实现安全管理的重要部分，通过创建一个新用户，并赋予一定的用户权限可以访问数据库资源，通过设置用户密码提供用户登录数据库的安全验证，Oracle 通过设置密码管理的概要文件来方便管理用户密码，通过资源管理的概要文件来赋予用户使用不同的数据库资源，如每个会话使用的 CPU 时间、每个会话的连接时间等，这些概要文件可以赋予不同的用户。根据业务需求，用户可以使用 ALTER PROFILE 指令更改概要文件的参数，或使用 DROP PROFILE 删除不需要的概要文件。

第 23 章

角 色 管 理

本章将讲解对角色的管理，角色是 Oracle 引入的权限的集合，通过将各种权限赋予角色，使得权限的赋予和回收非常方便，尤其是对于一个大的数据库系统中有很多不同的数据库用户，使用角色可以很好地减少 DBA 的用户权限管理。

23.1　什么是角色

角色是数据库各种权限的集合，使用角色可以方便地管理数据库特权，角色可以赋予其他用户，也可以赋予其他角色，图 23-1 形象地说明了角色的作用。

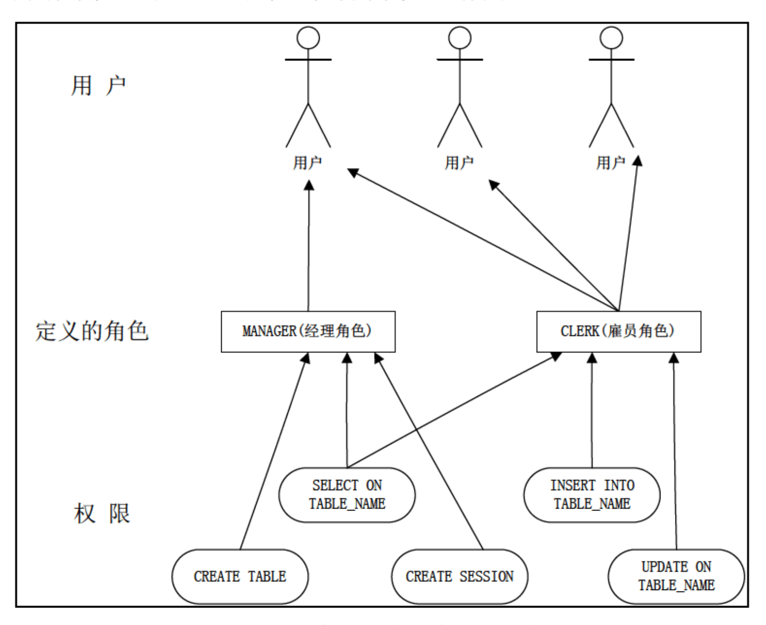


图 23-1　角色、用户和特权的关系图

显然，上图中有两个角色，而每个角色有不同数目和类型的权限，而这些角色又可以赋予不

同的用户，这样就方便了权限的管理，作为经理的角色（MANAGER）具有创建会话、创建表以及查询某个表的权限，而雇员（CLERK）角色具有查询表、更新表的权限。然后将 CLERK 角色和 MANAGER 角色赋予第一个用户，此时第一个用户就具有了图 23-1 中的所有权限，而把 CLERK 角色赋予第 2 和第 3 个用户。

使用角色可以减少给用户授予权限的操作次数，同样减少了修改多个用户的权限的操作次数，如系统上有 10 个用户，需要分别赋予这些用户 10 个权限，如果不使用角色，则需要 10*10=100 次操作，而如果使用角色将其作为 10 个特权的集合，再分别赋予 10 个用户，这样就需要 11 次操作，显然极大地减少了数据库操作，提高了 DBA 管理数据库的效率，并减少了出错机会。

下面总结一下角色的特点：

- 使用 GRANT 和 REVOKE 授予和回收权限。
- 可以授予任何用户或角色，但是不能赋予角色自己或循环赋予。
- 角色包含系统权限和对象权限。
- 允许启动或关闭赋予用户的角色。
- 允许使用密码启动一个角色。
- 角色名是唯一的，不能和已存在的用户名和角色名相同。
- 角色不被任何人拥有，也不属于任何模式。
- 角色的描述存储在数据字典 DBA_ROLES 中，如下所示。

```
SQL> select *
  2  from dba_roles;

ROLE                           PASSWORD
------------------------------------------------ -------------------
CONNECT                         NO
RESOURCE                        NO
DBA                             NO
SELECT_CATALOG_ROLE             NO
EXECUTE_CATALOG_ROLE            NO
............
```

既然角色有以上的特点，那么就从其特点出发总结角色的优点。

- 使得权限的管理更方便，角色赋予多个用户，使得相同的授权很容易实现，而如果需要修改这些用户的权限，只要修改角色就可以修改所有用户的权限。
- 动态的权限管理，一旦角色中的某个权限修改，则所有的被授予该角色的用户都自动获得修改的权限，并且立即生效。
- 权限可以激活和关闭，使得 DBA 可以方便地选择是否使用赋予用户的角色，临时的关闭或开启角色的使用。
- 可以通过操作系统授权角色，即角色可以通过操作系统指令或工具指定将角色赋予用户。
- 提高性能，使用角色减少了数据字典中授权记录的数量，通过关闭角色使得在语句执行过程中减少了权限的确认。

23.2　创建角色

在描述了什么是角色、总结了角色的优点之后，开始创建角色，创建角色的语法格式如下所示：

```
CREATE ROLE role name [NOT IDENTIFIED|IDENTIFIED {
BY password | EXTERNALLY | GLOBALLY | USING package}]
```

下面详细介绍各个参数的含义。

- role_name：角色名字。
- NOT IDENTIFIED：在激活角色时不需要密码验证。
- IDENTIFIED：在激活角色时需要密码验证。
- BY PASSWORD：设置激活角色的验证密码。
- USING package：创建应用角色，该角色只能由应用通过授权的 package 激活。
- EXTERNALLY：说明角色在激活前，必须通过外部服务，如操作系统或第三方服务授权。
- GLOBALLY：当使用 SET ROLE 激活角色时，用户必须通过企业路径服务授权来使用角色。

下面创建两个角色，注意此时的用户必须具有创建角色 CREATE ROLE 的权限，要创建的三个角色分别为 mk_clerk，at_clerk，manager，如下所示。

【实例 23-1】创建角色 mk_clerk，该角色不需要任何口令标识。

```
SQL> connect system/oracle@orcl
已连接。
SQL> create role mk_clerk;

角色已创建。
```

【实例 23-2】创建角色 at_clerk，该角色在激活时需要口令标识。

```
SQL> create role at_clerk
  2  identified by rmb;

角色已创建。
```

【实例 23-3】创建角色 manager，该角色使用外部服务来标识。

```
SQL> create role manager
  2  identified by externanlly;

角色已创建。
```

上面三个实例都成功创建了角色，那么如何查看创建的角色呢？Oracle 提供了数据字典 dba_roles，如实例 23-4 所示。

【实例 23-4】通过数据字典 dba_roles 来查看角色信息。

```
SQL> select *
  2  from dba_roles
  3  where role in ('MK_CLERK','AT_CLERK','MANAGER');
```

```
ROLE                           PASSWORD
------------------------------------------------------- --------
AT_CLERK                        YES
MANAGER                         YES
MK_CLERK                        NO
```

显然数据字典 dba_roles 仅记录了角色名和角色密码信息，角色 AT_CLERK 和角色 MANAGER 都需要密码，在 Oracle 9i 中角色 MANAGER 的 PASSWORD 属性为 EXTERNAL，在 Oracle 10g 中统一设置为 YES，只说明需要密码验证而不再细分密码类型。角色 MK_CLERK 在创建时没有使用 IDENTIFIED BY 选项，所以激活该角色时不需要密码验证。

23.3 修改角色

角色可以修改，但是 Oracle 只允许修改它的验证方法，修改角色的语法格式如下所示：

```
    ALTER ROLE role {NOT IDENTIFIED | IDENTIFIED {BY password | USING package |
EXTERNALLY | GLOBALLY}}
```

参数的含义如下所示，同创建角色时的参数含义相同，修改角色时只能修改验证方法：

- ROLE。
- NOT IDENTIFIED。
- BY password。
- EXTERNALLY。
- GLOBALLY。

下面修改在第 23.2 节创建的三个角色。

【实例 23-5】修改角色 MK_CLERK 的验证方法为外部标识。

```
SQL> alter role mk_clerk
  2  identified by externally;

角色已丢弃。
```

【实例 23-6】将角色 AT_CLERK 的验证方法改为不需要任何标识。

```
SQL> alter role at_clerk
  2  not identified;

角色已丢弃。
```

【实例 23-7】将角色 MANAGER 的验证方法改为需要密码标识。

```
SQL> alter role manager
  2  identified by rmb;

角色已丢弃。
```

上述三个实例已经成功修改了角色的验证方法，下面通过数据字典 DBA_ROLES 来验证修改结果，如实例 23-8 所示。

【实例 23-8】通过数据字典 DBA_ROLES 来验证角色的修改结果。

```
SQL> select *
  2  from dba_roles
  3  where role in ('MK_CLERK','AT_CLERK','MANAGER');

ROLE                          PASSWORD
------------------------------------------------ ------------------------
AT_CLERK                       NO
MANAGER                       YES
MK_CLERK                      YES
```

输出结果说明角色 AT_CLERK 不需要密码验证，而角色 MK_CLERK 和 MANAGER 需要密码验证，其中角色 MK_CLERK 需要外部服务方式验证，如通过操作系统验证等。

23.4　赋予角色权限

角色是权限的集合，所以在创建了角色后，就需要将各种权限赋予该角色，赋予角色权限的语法格式如下所示：

```
GRANT  权限 | 角色 TO  角色名
```

在使用该指令向角色授权时，可以将权限或角色赋予其他角色。

下面将 CREATE SESSION、SELECT TABLE、CREATE VEIW 权限赋予角色 AT_CLERK，如实例 23-9 所示。

【实例 23-9】为角色 CLERK 赋予权限。

```
SQL> grant create session,select any table,create view
  2  to at_clerk;

授权成功。
```

该例中将权限赋予了角色 AT_CLERK，通过数据字典 ROLE_SYS_PRIVS 验证授权结果，如实例 23-10 所示。

【实例 23-10】验证角色 AT_CLERK 的权限信息。

```
SQL> select *
  2  from role_sys_privs
  3  where role = 'AT_CLERK';

ROLE                          PRIVILEGE                    ADM
------------------------------------------------ ----------------
AT_CLERK                       CREATE SESSION                NO
AT_CLERK                       CREATE VIEW                  NO
```

```
AT_CLERK                          SELECT ANY TABLE                    NO
```

上述输出说明角色 AT_CLERK 具有了三个权限（PRIVILEGE），并且每个权限的 ADM 选项值都为 NO，说明该角色不能再将其拥有的权限赋予其他用户或角色。

下面将 CREATE ANY TABLE 的权限和角色 AT_CLERK 赋予角色 MANAGER。

【实例 23-11】将权限和角色 AT_CLERK 授予角色 MANAGER。

```
SQL> grant create any table,at clerk
  2  to manager;

授权成功。
```

同样通过数据字典 ROLE_SYS_PRIVS 检查角色 MANAGER 具有的权限，如实例 23-12 所示。

【实例 23-12】查看角色 MANAGER 具有的系统权限。

```
SQL> select *
  2  from role_sys_privs
  3  where role = 'MANAGER';

ROLE                             PRIVILEGE                        ADM
-------------------------------------------------- ------------------------
MANAGER                           CREATE ANY TABLE                NO
```

或许输出的结果令读者失望，命名授予了用户权限 CREATE ANY TABLE 和角色 AT_CLERK，但是没有角色 AT_CLERK 中的系统权限，这就说明使用该数据字典只能查询直接赋予该角色的权限，而无法查看授予该角色的角色信息。那么如何验证将角色 AT_CLERK 赋予角色 MANAGER 呢？Oracle 提供了一个数据字典 DBA_ROLE_PRIVS，如实例 23-13 所示。

【实例 23-13】通过数据字典 DBA_ROLE_PRIVS 查看角色授予信息。

```
SQL> select *
  2  from dba_role_privs
  3  where granted role = 'AT CLERK';

GRANTEE                     GRANTED_ROLE                ADM  DEF
--------------------------------------------- ------------------
MANAGER                           AT_CLERK              NO   YES
SYSTEM                            AT_CLERK               YES  YES
```

在上述输出中，说明角色 AT_CLERT 授予了用户 SYSTEM（该用户创建了角色）和角色 MANAGER。用户 SYSTEM 可以继续将该角色授予其他用户或角色，因为该行的 ADM 为 YES，并且角色 AT_CLERK 为用户 SYSTEM 的默认用户，因为该行的 DEF 为 YES，对于角色 MANAGER 而言，它不能再将角色 AT_CLERK 赋予其他用户或角色，因为该行的 ADM 为 NO。

23.5　赋予用户角色

在创建了角色后，就需要将各种角色赋予用户，或赋予所有用户（PUBLIC），将角色赋予用

户的语法格式如下所示：

```
GRANT role [, role ] ……
TO    {user | role | public} | [, { user | role | public }] ……
 [WITH ADMIN OPTION]
```

参数含义如下所示。

- role ：赋予用户的角色名（如多个角色，则用逗号隔开）。
- user：被赋予角色的用户（如多个用户用逗号隔开）。
- PUBLIC：将角色赋予所有用户。
- WITH ADMIN OPTION：被赋予该角色的用户或角色可以继续将该角色赋予其他用户或角色。

在将角色赋予用户前，先创建两个用户 clerk 和 manager，如下所示。

【实例 23-14】创建用户 clerk 和 manager。

```
SQL> create user clerk
  2  identified by cl12#;

用户已创建。
SQL> create user mymanager
  2  identified by my12#;

用户已创建。
```

通过数据字典 DBA_USERS 验证是否成功创建用户，如实例 23-15 所示。

【实例 23-15】验证是否成功创建用户。

```
SQL> select username,created
  2  from dba_users
  3* where username in ('CLERK','MYMANAGER')

USERNAME                          CREATED
------------------------------------------------ --------------
MYMANAGER                          17-9月 -09
CLERK                             17-9月 -09
```

上述输出说明已经创建了两个新用户 MYMANAGER 和 CLERK，下面将 MANAGER 角色赋予该用户。在赋予该用户之前先尝试通过用户 MYMANAGER 登录数据库，如实例 23-16 所示。

【实例 23-16】尝试使用新用户 MYMANAGER 登录数据库。

```
SQL> connect mymanager/my12#@orcl
ERROR:
ORA-01045: user MYMANAGER lacks CREATE SESSION privilege; logon denied


警告：您不再连接到 ORACLE。
```

提示说明用户 MYMANAGER 缺少 CREATE SESSION 权限，拒绝登录，这样的结果也是预料之中，因为新用户没有任何权利。下面将角色 MANAGER 赋予用户 MYMANAGER，并带有 WITH

ADMIN OPTION 选项。

【实例 23-17】将角色 MANAGER 赋予用户 MYMANAGER。

```
SQL> grant manager
  2* to mymanager with admin option

授权成功。
```

使用 DBA_ROLES_PRIVS 来验证授权结果，如实例 23-18 所示。

【实例 23-18】验证 MANAGER 角色授予的用户信息。

```
SQL> select *
  2  from dba_role_privs
  3  where granted_role ='MANAGER';

GRANTEE                        GRANTED_ROLE            ADM DEF
------------------------------------------------ ----------------------
MYMANAGER                      MANAGER                 YES YES
SYSTEM                          MANAGER                 YES YES
```

从输出可以清晰地看出角色 MANAGER 被赋予用户 MYMANAGER，并且该用户具有将角色 MANAGER 继续授权的能力，因为 ADM 值为 YES，角色 MANAGER 为用户 MANAGER 的默认角色，因为 DEF 值为 YES。

下面使用 MYMANAGER 用户登录数据库，如实例 23-19 所示。

【实例 23-19】使用被赋予 MANAGER 角色的用户 MYMANAGER 登录数据库。

```
SQL> connect mymanager/my12#@orcl
已连接。
```

成功连接数据库，说明角色 MANAGER 中的权限已起作用，下面查询当前用户 MYMANAGER 的当前会话级权限。

【实例 23-20】查看用户 MYMANAGER 的会话级权限。

```
SQL> select *
  2  from session_privs;

PRIVILEGE
----------------------------------------
CREATE SESSION
CREATE ANY TABLE
SELECT ANY TABLE
CREATE VIEW
```

显然这些权限全部是角色 MANAGER 中的权限，其中 CREATE ANY TABLE 系统权限是单独赋予角色 MANAGER 的，而其他三个权限是通过被赋予角色 AT_CLERK 而得到的。

在实例 23-17 中，赋予用户 MYMANAGER 角色时带有 WITH ADMIN OPTIN 选项，所以可以继续将角色 MANAGER 赋予其他用户，如实例 23-21 所示。

【实例 23-21】在 MYMANAGER 模式下将角色 MANAGER 赋予用户 CLERK。

```
SQL> grant manager
  2  to clerk;

授权成功。
```

提示授权成功，说明此时用户 CLERK 具有了角色 MANAGER 的所有权限。使用 CLERK 用户登录数据库。查询当前用户中角色的授予情况和当前用户的权限。

【实例 23-22】使用新用户 CLERK 登录数据库。

```
SQL> connect clerk/cl12#@orcl
已连接。
```

然后查询当前用户中角色的授予情况，如实例 23-23 所示。

【实例 23-23】查看 CLERK 用户的角色信息。

```
SQL> select *
  2  from user_role_privs;

USERNAME                          GRANTED_ROLE           ADM DEF OS_
-------------------------------------------------- -------------------
CLERK                              MANAGER                 NO  YES NO
```

输出说明用户 CLERK 被授予了角色 MANAGER，该角色不能被该用户继续授权，因为 ADM 的值为 NO，而且该角色不是操作系统验证的，因为参数 OS_为 NO，但是该角色是用户 CLERK 的默认角色，使用用户 CLERK 登录数据库时，该角色自动激活。

其实，读者可以预测此时用户 CLERK 具有同 MYMANAGER 用户一样的会话级权限。读者可以使用数据字典 SESSION_PRIVS 查询，如实例 23-20 所示，留作读者自己验证。

23.6　默认角色

在第 23.4 节，看到将角色 MANAGER 赋予用户 MYMANAGER 和用户 CLERK 后，角色 MANAGER 都是用户的默认角色，其实，这是一种默认设置。Oracle 允许使用 ALTER USER 指令修改默认角色。为了演示方便，下面将 AT_CLERK 角色赋予用户 CLERK。

【实例 23-24】将角色 AT_CLERK 赋予用户 CLERK。

```
SQL> connect system/oracle@orcl
已连接。
SQL> grant at_clerk to clerk;

授权成功。
```

使用数据字典 DBA_ROLE_RPIVES 查询用户 CLERK 被授予的角色信息。

【实例 23-25】查看用户 CLERK 被赋予的角色信息。

```
SQL> select *
```

```
  2  from dba_role_privs
  3  where grantee = 'CLERK';

GRANTEE                    GRANTED_ROLE          ADM DEF
------------------------------------------------- ------------------
CLERK                      MANAGER               NO  YES
CLERK                      AT_CLERK              NO  YES
```

我们看到用户 CLERK 被赋予了两个角色，即 MANAGER 和 AT_CLERK，而且都为默认角色。下面演示如何将角色 MANAGER 设置为非默认角色，如实例 23-26 所示。

【实例 23-26】将角色 MANAGER 设置为非默认角色。

```
SQL> alter user clerk default role all except manager;

用户已更改。
```

通过实例 23-27 验证修改结果，查看是否将角色 MANAGER 设置为非默认角色。

【实例 23-27】验证角色 MANAGER 是否为非默认角色。

```
SQL> select *
  2  from dba_role_privs
  3  where grantee = 'CLERK';

GRANTEE                    GRANTED_ROLE          ADM DEF
------------------------------------------------- --------------------
CLERK                      MANAGER               NO  NO
CLERK                      AT_CLERK              NO  YES
```

显然，用户 CLERK 的角色 MANAGER 不再是默认角色，因为 DEF 列的值为 NO。

如果读者需要把赋予该用户的角色全部设置为非默认角色，该怎么办？可以如实例 23-28 所示。

【实例 23-28】将用户 CLERK 的所有角色设置为非默认角色。

```
SQL> alter user clerk default role none;

用户已更改。
```

注意，此时用户 CLERK 不具有任何角色，也没有任何权利了。只有使用 SYSTEM 用户（其他具有相应访问权限的用户也可以）来登录数据库，查看用户 CLERK 的角色授予信息，如实例 23-29 所示。

【实例 23-29】使用 SYSTEM 用户登录数据库并查看用户 CLERK 的角色信息。

```
SQL> select *
  2  from dba_role_privs
  3  where grantee = 'CLERK';

GRANTEE                    GRANTED ROLE             ADM DEF
------------------------------------------------- --------------------
CLERK                      MANAGER                  NO  NO
CLERK                      AT_CLERK                 NO  NO
```

从输出可以看出用户 CLERK 具有两个角色，对应的 DEF 值都为 NO，说明在实例 23-28 中将所有角色设置为非默认角色的指令执行成功。

那么如何恢复角色 CLERK 中的其中一个、多个角色为默认角色呢？如实例 23-30 所示。

【实例 23-30】将用户 CLERK 的角色 AT_CLERK 设置为默认角色。

```
SQL> alter user clerk default role at_clerk ;

用户已更改。
```

此时，将角色 AT_CLERK 设置为用户 CLERK 的默认角色，使用数据字典 DBA_ROLE_PRIVS 查看修改情况。

【实例 23-31】验证是否将角色 AT_CLERK 设置为用户 CLERK 的默认角色。

```
SQL> select *
  2  from dba_role_privs
  3  where grantee = 'CLERK';

GRANTEE                           GRANTED_ROLE                 ADM  DEF
-------------------------------------------------  -------------------
CLERK                            MANAGER                        NO   NO
CLERK                            AT_CLERK                        NO   YES
```

显然，输出结果中角色 AT_CLERK 对应的行（第 2 行）属性 DEF 的值为 YES，说明将角色 AT_CLERK 设置为用户 CLERK 的默认角色修改成功。

如果要把授予用户的多个角色都设置为默认角色，可以使用如下的方式实现。仍然以 CLERK 用户为例，如实例 23-32 所示。

【实例 23-32】将赋予用户的所有角色设置为默认角色。

```
SQL> connect system/oracle@orcl
已连接。
SQL> ALTER USER clerk DEFAULT ROLE ALL;

用户已更改。
```

现在输出提示“用户已更改”，说明已经成功将用户 clerk 的所有角色都设置为默认角色。为了验证这个结果，再次使用数据字典 dba_role_privs 来查看用户 clerk 的角色信息。

【实例 23-33】再次查看用户信息。

```
SQL> select *
  2  from dba_role_privs
  3  where grantee ='CLERK';

GRANTEE                           GRANTED_ROLE               ADM  DEF
-------------------------------------------------  ----------------------
CLERK                            MANAGER                    NO   YES
CLERK                            AT_CLERK                    NO   YES
```

显然，此时角色 MANAGER 也为用户 CLERK 的默认角色了，因为 DEF 值为 YES，使用这种

方式设置用户角色为默认角色很方便，尤其都有多个角色的用户而言更是如此。



在将某个角色设置为用户默认角色时，如果该角色是通过其他角色授予该用户的，则该角色不能在 DEFAULT ROLE 子句中使用。

23.7　禁止和激活角色

角色可以禁止和激活，禁止意味着用户不再具有该角色赋予的各种权限，即回收角色具有的权限，而激活意味着赋予用户角色的权限。在实例 23-33 中，用户 CLERK 具有两个角色 MANAGER 和 AT_CLERK，下面查看该用户具有的角色权限，如实例 23-34 所示。

【实例 23-34】查看用户 CLERK 的用户权限。

```
SQL> connect clerk/cl12#@orcl
已连接。
SQL> select *
  2  from session privs;

PRIVILEGE
----------------------------------------
CREATE SESSION
CREATE ANY TABLE
SELECT ANY TABLE
CREATE VIEW
```

这里再说明一下角色 MANAGER 的权限，该角色通过被赋予 CREATE ANY TABLE 权限和角色 CLERK 而获得权限，而角色 CLERK 通过被赋予 CREATE SESSION、SELECT ANY TABLE、CREATE VIEW 来获得权限。这样用户 CLERK 被赋予了两个角色 MANAGER 和 CLERK，这样它就具有了 4 个权限。

下面演示如何禁止用户的角色，如实例 23-35 所示，禁止用户 CLERK 的所有角色。

【实例 23-35】禁止用户的所有角色。

```
SQL> set role none;

角色集
```

此时，系统禁止了用户的所有角色，即系统回收了这些角色的权限。查询当前用户 CLERK 的权限如实例 23-36 所示。

【实例 23-36】在禁止所有角色后查询用户权限。

```
SQL> select *
  2  from session_privs;

未选定行
```

可见，用户 CLERK 不具有任何会话权限了，下面激活用户 CLERK 的 AT_CLERK 角色。

【实例 23-37】激活用户 CLERK 的角色 AT_CLERK。

```
SQL> SET ROLE at_clerk ;

角色集
```

激活了角色 AT_CLERK 后查询用户 CLERK 是否具有了角色 AT_CLERK 的权限。

【实例 23-38】查询用户 CLERK 是否具有角色 AT_CLERK 的权限。

```
SQL> select *
  2  from session_privs;

PRIVILEGE
----------------------------------------
CREATE SESSION
SELECT ANY TABLE
CREATE VIEW
```

该输出和实例 23-36 不同，这里用户 CLERK 具有了角色 AT_CLERK 的权限。下面再激活角色 MANAGER。

【实例 23-39】激活角色 MANAGER。

```
SQL> set role manager;
set role manager
*
第 1 行出现错误:
ORA-01979: 角色 'MANAGER' 的口令缺失或无效
```

没有成功激活，错误提示为缺少口令，读者或许有印象，在创建角色 MANAGER 时使用了验证方式，即使用密码验证。再次通过密码验证的方式激活角色 MANAGER。

【实例 23-40】激活设置了密码验证的角色 MANAGER。

```
SQL> set role manager identified by rmb;

角色集
```

此时，已激活角色 MANAGER，通过数据字典 SESSION_PRIVS 查看用户 CLERK 是否具有了 CREATE ANY TABLE 权限（该权限是角色 MANAGER 拥有的），如实例 23-41 所示。

【实例 23-41】查看用户 CLERK 的权限。

```
SQL> select *
  2  from session_privs;

PRIVILEGE
-------------------------------
CREATE SESSION
CREATE ANY TABLE
SELECT ANY TABLE
CREATE VIEW
```

从输出可以看出用户 CLERK 具有角色 MANAGER 和角色 CLERK 的权限，说明成功激活了角色 MANAGER。

23.8 回收和删除角色

既然可以赋予用户角色，也可以回收用户角色，Oracle 允许使用 REVOKE 子句回收赋予某一用户的角色。为了演示如何回收角色，下面在系统上创建的角色和查看这些角色赋予的用户信息等。

【实例 23-42】查看创建的角色信息。

```
SQL> connect system/oracle@orcl
已连接。
SQL> select *
  2  from dba_roles
  3  where role in ('AT_CLERK','MANAGER');

ROLE                           PASSWORD
------------------------------------------------ ------------------
AT CLERK                        NO
MANAGER                        YES
```

创建的角色 AT_CLERK 和 MANAGER 已存在，然后继续查询这两个角色赋予的用户信息，即哪些用户被赋予了这些角色，如实例 23-43 所示。

【实例 23-43】查看角色 AT_CLERK 和 MANAGER 赋予的用户信息。

```
SQL> select *
  2  from dba role privs
  3  where granted_role in  ('AT_CLERK','MANAGER')
  4  order by granted_role;

GRANTEE                         GRANTED ROLE            ADM DEF
-------------------------------------------------- ---------------------
CLERK                           AT_CLERK                  NO  YES
MANAGER                        AT CLERK                 NO  YES
SYSTEM                          AT_CLERK                 YES  YES
MYMANAGER                     MANAGER                 YES  YES
SYSTEM                          MANAGER                  YES YES

已选择 6 行。
```

从输出可以看出角色 AT_CLERK 赋予了用户 CLERK、MANAGER 和 SYSTEM，而角色 MANAGER 赋予了用户 MYMANAGER 和用户 SYSTEM。注意，角色 AT_CLERK 和 MANAGER 是在用户 SYSTEM 下创建的，所以该用户自动被授予该角色，并且为默认角色。

下面回收用户 CLERK 的 AT_CLERK 角色，如实例 23-44 所示。

【实例 23-44】回收用户 CLERK 的 AT_CLERK 角色。

```
SQL> revoke at_clerk from clerk;

撤销成功。
```

然后，检验角色 AT_CLERK 是否已经从用户 CLERK 回收，如实例 23-45 所示。

【实例 23-45】验证是否正确回收用户 CLERK 的角色 AT_CLERK。

```
SQL> select *
  2  from dba_role_privs
  3  where granted_role = 'AT_CLERK'
  4  ;

GRANTEE                        GRANTED_ROLE                   ADM DEF
------------------------------ ------------------------------ --- ---
MANAGER                        AT_CLERK                       NO  YES
SYSTEM                          AT_CLERK                        YES YES
```

在上述查询的 WHERE 子句中，限制查询 GRANTED_ROLE 为 AT_CLERK 的用户或角色信息，在 GRANTEE 列中没有了用户 CLERK，说明角色 AT_CLERK 已经从用户 CLERK 回收。



在回收角色时，可以同时回收多个角色，角色名之间使用逗号隔开，也可以同时从几个用户回收一个或多个相同的角色，用户名之间使用逗号隔开。

但是角色 AT_CLERK 依旧在系统中存在，只是没有授予用户使用，如果不需要该角色可以删除该角色，如实例 23-46 所示。

【实例 23-46】删除角色 AT_CLERK。

```
SQL> drop role at_clerk;

角色已删除。
```

通过数据字典 DBA_ROLES 查看是否成功删除角色 AT_CLERK，如实例 23-47 所示。

【实例 23-47】验证角色 AT_CLERK 是否删除。

```
SQL> select *
  2  from dba roles
  3  where role in ('AT_CLERK','MANAGER');

ROLE                           PASSWORD
-------------------------------------------------- ------------------
MANAGER                        YES
```

显然，ROLE 列中没有了角色 AT_CLERK，说明成功删除角色 AT_CLERK。有这样一种情况就是需要将一个角色授予所有的用户，如查询普通的表权限的角色，如果将这样的角色赋予所有的用户，也是很耗时的，Oracle 使用 PUBLIC 代表所有用户，可以使用如下的方式将一个或多个角色赋予所有用户，如实例 23-48 所示。

【实例 23-48】将角色授予所有用户。

```
SQL> grant manager to public;

授权成功。
```

此时，将 MANAGER 角色赋予了所有用户，然后通过数据字典 DBA_ROLE_PRIVS 查看角色 MANAGER 的赋予用户信息。

【实例 23-49】查看角色 MANAGER 的赋予用户信息。

```
SQL> select *
  2  from dba_role_privs
  3* where granted_role in ('MANAGER','PUBLIC')

GRANTEE                            GRANTED_ROLE                ADM DEF
------------------------------------------------- -------------------------
MYMANAGER                          MANAGER                    YES YES
SYSTEM                             MANAGER                    YES YES
PUBLIC                             MANAGER                    NO  YES
```

此时，我们看到角色 MANAGER 赋予了三个用户，即 MYMANAGER、SYSTEM 和 PUBLIC。那么如何回收授予 PUBLIC 的角色呢？如实例 23-50 所示。

【实例 23-50】回收授予 PUBLIC 的 MANAGER 角色。

```
SQL> revoke manager from public;

撤销成功。
```

为了验证角色的回收结果，下面查询是否成功回收授予 PUBLIC 的角色 MANAGER。

【实例 23-51】查询是否成功回收授予 PUBLIC 的角色 MANAGER。

```
SQL> select *
  2  from dba role privs
  3* where granted_role in ('MANAGER','PUBLIC')

GRANTEE                            GRANTED_ROLE                ADM DEF
------------------------------------------------- -------------------------
MYMANAGER                          MANAGER                    YES YES
SYSTEM                             MANAGER                    YES YES
```

显然，已经成功从 PUBLIC 回收角色 MANAGER，因为 GRANTEE 列已经没有 PUBLIC 值，但是单独授予用户的 MANAGER 角色不会通过 REVOKE MANAGER FROM PUBLIC 而被回收，如授予用户 MYMANAGER 的角色 MANAGER 就继续保留下来。

23.9　预定义角色

除了自定义的角色外，Oracle 还预定义了一些很有用的角色，如 DBA 角色、RESOURCE 角

色等。下面列出这些角色，并分析这些角色的作用。

Oracle 定义的角色列表（这是 Oracle10g 中系统预定义的角色）如下。

- AQ_ADMINISTRATOR_ROLE：QUEUE 的管理员角色。
- CONNECT：连接数据库权限。
- DBA：数据库管理员权限。
- EXP_FULL_DATABASE：导出数据库权限。
- IMP_FULL_DATABASE：导入数据库权限。
- JAVADEBUGPRIV：调试 Java 程序权限。
- MGMT_USER：创建会话和创建触发器权限。
- OEM_ADVISOR：执行 OEM 顾问的权限。
- OEM_MONITOR：执行 OEM 监视的权限。
- OLAP_DBA：执行联机事务处理时的 DBA 权限。
- OLAP_USER：执行联机事务处理时的 USER 权限。
- RESOURCE：创建一系列数据库对象的权限。
- SCHEDULER_ADMIN：管理各种调度的权限，如创建任务、执行程序等。

在 Oracle 10g 中对于预定义的角色安全性有一定程度的加强。在 Oracle 9i 中，CONNECT 角色具有创建视图、表、建立会话、创建同义词、序列号以及创建数据库连接的权限，而在 Oracle 10g 中该角色只具有 CREATE SESSION 权限，显然提高了系统的安全性。并且在 Oracle 10g 中去除了 RESOURCE 角色，而 Oracle 9i 中 RESOURCE 角色和 CONNECT 角色的部分权限统一放入 DBA 角色中。

读者可以通过数据字典 ROLE_SYS_PRIVS 查询系统的预定义角色，可先使用 SYSTEM 用户进行登录，如实例 23-52 所示。

【实例 23-52】使用 SYSTEM 用户登录数据库。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
```

然后，查询 EXP_FULL_DATABASE 角色有哪些权限，如实例 23-53 所示。

【实例 23-53】查询角色 EXP_FULL_DATABASE 的权限信息。

```
SQL> select *
  2  from role_sys_privs
  3  where role ='EXP_FULL_DATABASE'
  4* order by privilege

ROLE                          PRIVILEGE                          ADM
---------------------------------------------- --------------------
EXP_FULL_DATABASE             ADMINISTER RESOURCE MANAGER      NO
EXP_FULL_DATABASE             BACKUP ANY TABLE                  NO
EXP_FULL_DATABASE             EXECUTE ANY PROCEDURE             NO
EXP_FULL_DATABASE             EXECUTE ANY TYPE                   NO
EXP_FULL_DATABASE             READ ANY FILE GROUP                NO
```

```
EXP_FULL_DATABASE          RESUMABLE                         NO
EXP_FULL_DATABASE          SELECT ANY SEQUENCE               NO
EXP_FULL_DATABASE          SELECT ANY TABLE                  NO

已选择 8 行。
```

从 PRIVILEGE 列的值可以清楚地看出角色 EXP_FULL_DATABASE 的权限，但是该角色一旦授予用户，该用户不能继续将该角色授予其他用户或角色，因为 ADM 列的值为 NO。

23.10　本章小结

本章讲解了角色管理，角色就是数据库权限的集合，这些权限可以是系统权限，也可以是数据库对象权限，使用角色可以方便地对用户授权进行管理，并减少授权操作。用户可以根据业务需求来设计角色所拥有的权限，再将这些权限使用 grant 子句赋予不同的用户。赋予用户的权限在不需要的时候可以使用 REVOKE 子句回收角色，也可以禁止和激活赋予用户的角色，此时使用 SET ROLE NONE 来禁止所有的角色，使用 SET ROLE role_name 来激活角色。如果角色在定义时使用了密码验证，可使用 SET ROLE role_name INDENTIFIED BY password 的方式激活该角色。

为了方便管理 Oracle 预定义的一些角色，如 CONNECT、DBA 角色等，读者可以通过数据字典 ROLE_SYS_PRIVS 查询这些预定义角色，以及这些角色的权限。

第 24 章

系统和对象权限管理

权限管理是 Oracle 实现安全管理的一部分，通过授予不同用户的系统权限和对象权限实现用户对系统功能以及数据库对象的操作，本章分别讲解系统权限和对象权限，并通过实例说明如何授予和回收对象权限和系统权限。

权限是执行特殊 SQL 语句或访问其他用户拥有的对象的权利，这些权利包括：连接数据库、维护表空间、改变系统状态、创建表、从用户表中选择数据行、执行存储过程等。

Oracle 将权限分为系统权限和对象权限。

- 系统权限：系统权限允许用户执行一个或一类特殊的数据库操作，如创建数据库、创建用户、创建与维护表空间以及管理会话等。
- 对象权限：对象权限是用户维护数据库对象的权利，如维护表、视图、序列号、存储过程、函数等。

24.1　系统权限

在 Oracle 数据库中有一类具有最高权限的用户，它可以实现对数据维护的任何工作，即 DBA 用户。DBA 用户可以为新用户或其他用户授权以执行某种操作，如赋予 SCOTT 用户访问所有对象的表的权利、赋予或回收执行系统功能的权利、直接将权限赋予用户或角色以及将权限赋予所有用户（PLUBLIC）。

24.1.1　什么是系统权限

在 Oracle 数据库中有 100 多种系统权限，在权限中的 ANY 关键字说明在任何模式中当前被授权的用户都具有这种权限，如 SELECT ANY TABLE 说明可以选择任何模式对象，GRANT 指令向用户或一组用户赋予某种权限，如 GRANT SELECT ANY TABLE TO SCOTT，就是向用户 SCOTT 赋予查看任何表的权限，而 REVOKE 指令说明要删除某个特权，如 REVOKE SELECT ANY TABLE FROM SCOTT，就是从 SCOTT 用户回收查看所有表的权限。

下面介绍常用的几类系统权限。

1. 与索引相关的系统权限

系统权限如下。

- CREATE ANY INDEX：创建任何模式中对象的索引。
- ALTER ANY INDEX：修改任何模式的索引。
- DROP ANY INDEX：删除任何模式的索引。



没有 CREATE INDEX 的权限。在 CREATE TABLE 权限中包含了 CREATE INDEX 的权限。

2. 与表相关的权限

与表相关的权限如下。

- CREATE TABLE：在当前模式中创建表。
- CREATE ANY TABLE：在任何模式中创建表。
- ALTER ANY TABLE：修改任何模式中的表。
- DROP ANY TABLE：删除任何模式中的表。
- SELECT ANY TABLE：查看任何模式中的表数据。
- UPDATE ANY TABLE：修改任何模式中的表数据。
- DELETE ANY TABLE：删除任何模式中的表。



具有 CREATE TABLE 权限的用户自然拥有删除该表对象的权限，其他与 CREATE PROCEDURE、CREATE FUNCTION 类似。在创建表时必须为表分配表空间配额，或者对表空间具有无限制使用权限，如 UNLIMITED TABLESPACE。

3. 与会话相关的权限

与会话相关的权限如下。

- CREATE SESSION：建立数据库会话的权限。
- ALTER SESSION：修改数据库会话的权限。



一个新用户创建后，首先需要授予 CREATE SESSION 权限，它可以访问数据库。

4. 与表空间相关的权限

与表空间相关的权限如下。

- CREATE TABLESPACE：创建表空间。
- ALTER TABLESPACE：修改表空间。

- DROP TABLESPACE：删除表空间。
- UNLIMITED TABLESPACE：允许使用所有表空间的权限。

24.1.2 授予系统权限

要授予用户系统权限需要使用 GRANT 指令的 SQL 语句，被授予了权限的用户在一定授权下可以继续将系统权限赋予其他用户。下面是赋予用户权限的语法格式。

```
GRANT { system privilege | role }
        [, { system_privilege | role } ] ……
   TO    { user | role | PUBLIC }
        [, { user | role | PUBLIC } ] ……
     [ WITH ADMIN OPTION ]
```

如上所示整个授权的框架是：GRANT 系统权限 TO 用户 [WITH ADMIN OPTION]。如果有多个系统权限使用逗号隔开，如果有多个用户也用逗号隔开。在上述语法格式中符号“|”表示“或”的关系。

下面通过实例演示如何向用户授权。

首先，创建一个用户 JNAE。

【实例 24-1】创建新用户 JANE。

```
SQL> create user jane
  2  identified by abc124#;

用户已创建。
```



上例中的密码有点奇怪，因为我们使用了 verify_function 密码复杂性验证函数，且更改了用户的默认密码概要文件，所以这里的密码要符合设置的规则。

对于一个新创建的数据库用户，它不具有任何权限，如果此时使用该用户连接数据库则无法成功。

【实例 24-2】使用新用户 JANE 连接数据库。

```
SQL> connect jane/abc124#@orcl
ERROR:
ORA-01045: user JANE lacks CREATE SESSION privilege; logon denied


警告：您不再连接到 ORACLE。
```

该实例说明用户 JNAE 没有 CREATE SESSION 的系统权限，所以登录被拒绝了。下面将 CREATE SESSION、CREATE TABLE、SELECT ANY TABLE 的权限赋予用户，此时必须以 SYSTEM 用户登录。

【实例 24-3】赋予用户权限。

```
SQL> grant create session,create table,select any table to jane;
```

```
授权成功。
```

现在用户 JANE 具有了建立数据库会话、创建表以及查看任何表的系统权限，下面通过数据字典 DBA_SYS_PRIVS 查看被授权的用户的权限信息。

【实例 24-4】查看用户 JANE 拥有的系统权限。

```
SQL> conn system/oracle@orcl
已连接。
SQL> col grantee for a10
SQL> col privilege for a25
SQL> select *
  2  from dba_sys_privs
  3* where grantee = 'JANE'

GRANTEE    PRIVILEGE                 ADM
------------------- ---------------------------------------- ----------
JANE       SELECT ANY TABLE          NO
JANE       CREATE TABLE              NO
JANE       CREATE SESSION            NO
```

从输出可以看出用户 JANE 具有了三个系统权限，即 SELECT ANY TABLE、CREATE TABLE、CREATE SESSION。而 ADM 列的值都为 NO，说明用户 JANE 拥有的权限不能再赋予其他用户。为了验证整个问题，我们再创建一个新用户 LARRY。

【实例 24-5】创建用户 LARRY。

```
SQL> create user larry
  2  identified by abc124#;

用户已创建。
```

新用户创建成功，我们使用数据字典 DBA_SYS_PRIVS 查看该用户的系统权利是不是“一穷二白”。

【实例 24-6】查看用户 LARRY 的系统权限。

```
SQL> select *
  2  from dba_sys_privs
  3  where grantee = 'LARRY';

未选定行
```

显然，用户 LARRY 目前还不具有任何系统权限，下面尝试使用 JANE 用户登录并将 CREATE SESSION、SELECT ANY TABLE 的权限赋予用户 LARRY。

【实例 24-7】用户 JANE 赋予用户 LARRY 系统权限。

```
SQL> connect jane/abc124#@orcl
已连接。
SQL> grant create session,select any table to larry;
grant create session,select any table to larry
*
```

```
第 1 行出现错误:
ORA-01031: 权限不足
```

发生错误，说明权限不够，用户 JANE 无法向用户 LARRY 授权。读者还记得在实例 24-4 中用户 JANE 的 ADM 列的值都为 NO，所以无法继续授权给其他用户。下面修改用户 JANE 的权限使得它具有继续授权给其他用户的权利，先回收用户权限，与实例 24-8 所示。

【实例 24-8】回收用户 JANE 的所有权限。

```
SQL> connect system/oracle@orcl
已连接。
SQL> revoke create session,select any table,create table from jane;

撤销成功。
```

下面重新授予用户 JANE 这些权限，并带有 WITH ADMIN OPTION 选项，如实例 24-9 所示。

【实例 24-9】赋予用户 JANE 系统权限并允许继续授权。

```
SQL> grant create session,select any table,create table to jane
  2  with admin option;

授权成功。
```

为了验证 WITH ADMIN OPTION 的参数设置效果，继续使用数据字典 DBA_SYS_PRIVS。

【实例 24-10】查看用户 JANE 的系统权限信息。

```
SQL> select *
  2  from dba_sys_privs
  3  where grantee ='JANE';

GRANTEE    PRIVILEGE                 ADM
-------------------- ---------------------------------------- -----------
JANE       SELECT ANY TABLE          YES
JANE       CREATE TABLE              YES
JANE       CREATE SESSION            YES
```

此时用户 JANE 系统权限的 ADM 列的值都为 YES，说明这些权限可以继续赋予其他用户。下面将用户 JANE 的 CREATE SESSION 和 SELECT ANY TABLE 赋予用户 LARRY。

【实例 24-11】用户 JANE 赋予用户 LARRY 系统权限。

```
SQL> connect jane/abc124#@orcl
已连接。
SQL> grant create session,select any table to larry;

授权成功。
```

授权成功，通过数据字典 DBA_SYS_PRIVS 查看用户 LARRY 具有的系统权限。

【实例 24-12】查看用户 LARRY 的系统权限。

```
SQL> connect system/oracle@orcl
```

```
已连接。
SQL> select *
  2  from dba_sys_privs
  3  where grantee ='LARRY';

GRANTEE    PRIVILEGE                   ADM
-------------------- ------------------------------------- ----------
LARRY      CREATE SESSION              NO
LARRY      SELECT ANY TABLE            NO
```

从输出可以清楚地看出用户 LARRY 具有了 CREATE SESSION 和 SELECT ANY TABLE 的权限，但是用户 LARRY 不能将这些权限再赋予其他用户，因为在向用户 LARRY 授权时，没有使用 WITH ADMIN OPTION 选项。

下面使用 LARRY 用户登录数据库，并查询 SCOTT 用户的表信息。

【实例 24-13】使用用户 LARRY 登录数据库。

```
SQL> connect larry/abc124#@orcl
已连接。
SQL> select *
  2  from scott.dept;

   DEPTNO DNAME          LOC
------------------------------------------ --------------
       40 OPERATION       BOSTON
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
```

因为用户 LARRY 被赋予了 CREATE SESSION 的权限，所以可以成功连接数据库，而 SELECT ANY TABLE 的权限使得它可以查看任何用户的表信息。

因为创建的用户都是用于连接数据库的，并查看一些表信息，如果业务允许我们事先将一些权限赋予当前所有的用户，如 CREATE SESSION、SELECT ANY TABLE 等，如实例 24-14 所示。

【实例 24-14】将部分系统权限赋予所有用户。

```
SQL> conn system/oracle@orcl
已连接。
SQL> grant create session,select any table to public;

授权成功。
```

Oracle 提供了两个特殊的系统权限，即 SYSDB 权限和 SYSOPER 权限，在做系统维护时建议使用这两种系统权限登录数据库。用户通过 SYSDBA 连接到数据库时，它具有对数据库的一切特权。下面是两种系统特权的典型数据库操作。

与 SYSDBA 系统特权相关的操作。

- SYSOPER PRIVILEGES WITH ADMIN OPTION：具有 SYSOPER 所具有的操作，并且可以将这些特权赋予其他用户。

- CREATE DATABASE：创建数据库。
- ALTER DATABASE BEGIN/END BACKUP：将数据库置于备份状态。
- RESTRICTED SESSION：设置会话限制。
- RECOVER DATABASE UNTIL：介质恢复数据库到 UNTIL 指定的状态。

【实例 24-15】使用 SYSDBA 特权登录数据库。

```
SQL> connect system/oracle@orcl as sysdba;
已连接。
```

与 SYSOPER 系统特权相关的操作。

- STARTUP：启动数据库。
- SHUTDOWN：关闭数据库。
- ALTER DATABASE OPEN | MOUNT：将数据库切换到打开|挂起状态。
- ALTER DATABASE BACKUP CONTROLFILE TO：备份控制文件。
- RECOVER DATABASE：介质恢复数据库。
- ALTER DATABASE ARCHIVELOG：将数据库设置为归档模式。

24.1.3　回收系统权限

如果需要限制某个用户的权限可以回收权限，使用 REVOKE 指令，回收用户权限的语法格式如下所示。

```
REVOKE {system_privilege | role }
     [, {system privilege | role ] ……
FROM { user | role | PUBLIC }
     [,{user | role | PUBLIC} ] ……
```

下面查询用户 JANE 和 LARRY 的用户系统权限，如实例 24-16 所示。

【实例 24-16】查询用户 JANE 和 LARRY 的系统权限。

```
SQL> col grantee for a10
SQL> col privilege for a25
SQL> col admin_option for a15
SQL> select *
  2  from dba sys privs
  3  where grantee IN ('JANE','LARRY')
  4* order by grantee

GRANTEE    PRIVILEGE                ADMIN_OPTION
------------------- --------------------------------------- ----
JANE       CREATE SESSION           YES
JANE       CREATE TABLE             YES
JANE       SELECT ANY TABLE         YES
LARRY      CREATE SESSION           NO
LARRY      SELECT ANY TABLE         NO
```

从以上输出可以看出，用户 JANE 和 LARRY 都具有系统权限，下面演示如何回收用户 LARRY

的所有系统权限，如实例 24-17 所示。

【实例 24-17】回收用户 LARRY 的系统权限。

```
SQL> connect system/oracle@orcl
已连接。
SQL> revoke create session,select any table
  2  from larry;

撤销成功。
```

撤销成功说明回收了用户 LARRY 的系统权限，下面验证回收结果。

【实例 24-18】查询用户 LARRY 的系统权限。

```
SQL> select *
  2  from dba_sys_privs
  3  where grantee ='LARRY';

未选定行
```

输出结果说明，数据字典 DBA_SYS_PRIVS 中没有记录用户 LARRY 的系统权限信息，说明实例 24-17 已成功回收用户 LARRY 的系统权限。

对于授权时使用了 WITH ADMIN OPTION 选项的用户的权限回收工作需要做一些说明，所以再次为用户 LARRY 赋予 CREATE SESSION 和 SELECT ANY TABLE 的权限，并带 WITH ADMIN OPTION 选项，创建一个新用户 SOPHIE。

【实例 24-19】为用户 LARRY 赋予系统权限并带 WITH ADMIN OPTION 选项。

```
SQL> grant create session,select any table
  2  to larry with admin option;

授权成功。
```

【实例 24-20】创建新用户 SOPHIE。

```
SQL> create user sophie
  2  identified by abc124#;

用户已创建。
```

此时使用 LARRY 用户登录数据库，然后将 CREATE SESSION 和 SELECT ANY TABLE 的权限赋予用户 sophie，如实例 24-21 所示。

【实例 24-21】用户 LARRY 向用户 SOPHIE 授予系统权限。

```
SQL> conn larry/abc124#@orcl
已连接。
SQL> grant create session,select any table
  2  to sophie;

授权成功。
```

此时，用户 SOPHIE 具有 CREATE SESSION 和 SELECT ANY TABLE 的系统权限，为了验证

结果，再次使用数据字典 DBA_SYS_PRIVS，如实例 24-22 所示。

【实例 24-22】查询用户 SOPHIE 的系统权限信息。

```
SQL> conn system/oracle@orcl
已连接。
SQL> col grantee for a10
SQL> col privilege for a25
SQL> col admin option for a15
SQL> select *
  2  from dba_sys_privs
  3* where grantee ='SOPHIE'

GRANTEE    PRIVILEGE                ADMIN_OPTION
------------------ ---------------------------------------- ---
SOPHIE     CREATE SESSION            NO
SOPHIE     SELECT ANY TABLE          NO
```

此时，用户 JANE 向用户 LARRY 授予系统权限，用户 LARRY 具有继续向其他用户授予系统权限的能力，此时用户 LARRY 又向新用户 SOPHIE 授予系统权限，那么如果用户 JANE 回收了用户 LARRY 的 SELECT ANY TABLE 的权利，是否影响用户 SOPHIE 的 SELECT ANY TABLE 的系统权限呢？通过实例 24-23 进行说明。

【实例 24-23】用户 JANE 回收用户 LARRY 的 SELECT ANY TABLE 权限。

```
SQL> connect jane/abc124#@orcl
已连接。
SQL> revoke select any table
  2  from larry;

撤销成功。
```

输出显示已成功回收用户 LARRY 的 SELECT ANY TABLE 系统权限，接着验证用户 SOPHIE 是否还具有 SELECT ANY TABLE 权限，使用用户 SOPHIE 登录数据库，然后查询表信息。

【实例 24-24】使用用户 SOPHIE 登录数据库并查询表信息。

```
SQL> connect sophie/abc124#@orcl
已连接。
SQL> select *
  2  from scott.dept;

   DEPTNO DNAME          LOC
--------------- ---------------------------- -------------
       40 OPERATION      BOSTON
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
```

显然，虽然用户 LARRY 的 SELECT ANY TABLE 系统权限被回收了，但是用户 SOPHIE 仍然具有 SELECT ANY TABLE 权限。

上例与 WITH ADMIN OPTION 选项相关的权限回收示例说明了 REVOKE 回收系统权限不具备级联特性。

回收用户的系统权限，如实例 24-25 所示。

【实例 24-25】回收授予所有用户的系统权限。

```
SQL> connect system/oracle@orcl
已连接。
SQL> revoke create session,select any table
  2  from public;

撤销成功。
```

24.2 对象权限

和系统权限相对应的是对象权限，对象包括表、视图（物化视图）、序列号和存储过程等。

24.2.1 什么是对象权限

在这些数据库对象上实现某种特殊的行为的权限称为对象权限，如对于一个表可以更改表的结构、删除表或更新表中的数据行等。Oracle 的对象权限包括：ALTER、DELETE、EXECUTE、INDEX、INSERT、REFERENCES、SELECT 和 UPDATE。而这些对象权限适用于不同的数据库对象。

表 24-1 列举了数据库对象的权限与对应的数据库对象的关系。

表 24-1 对象权限列表

对象权限	表（table）	视图（view）	序列号（sequence）	过程（procedure）
ALTER	y		y	y
DELETE	y	y		
EXECUTE				y
INDEX	y	y		
INSERT	y	y		
REFERENCES	y			
SELECT	y	y	y	
UPDATE	y	y		

在上述对象的权限列表中，SEQUENCE 序列号只有两种对象权限，即 ALTER 和 EXECUTE。而对于对象权限 ALTER、UPDATE、REFEREMCES 和 INSERT 可以实现更小粒度的权限控制，如可以对表对象的某列加以限制等。

在给出一个具体的对象权限授予的实例前，先给出对象授权的语法格式：

```
GRANT {object_privilege [ ( column_list ) ]
     [, object privilege [ ( column list) ] ] ……
     | ALL [ PRIVILEGE ]
```

```
ON   [schema. ] object
TO   { user | role | PUBLIC } [, { user | role |PUBLIC } ]……
     [ WITH GRANT OPTION]
```

对上述命令的说明如下。

- GRANT：授权关键字。
- Object_privilege：对象权限
- Column_list：对象权限操作的列表。
- All：将当前用户的某个数据库对象的所有权限赋予新用户。
- On object：说明具体的数据库对象，如表或存储过程。
- With grant option：新用户可以继续授权。

接下来，给出一个将表对象权限 UPDATE 赋予新用户 LARRY 的实例，并且 LARRY 用户可以继续将该对象权限赋予其他用户。

【实例 24-26】把 SCOTT 用户的 EMP 表权限 UPDATE 赋予新用户 LARRY。

```
SQL> connect scott/tiger@bjyzz
已连接。
SQL> grant update on emp
  2  to larry with grant option;

授权成功。
```

上例中，使用 SCOTT 用户登录数据库，并且将 SCOTT 用户 EMP 表的 UPDATE 权限赋予用户 LARRY。下面通过数据字典 USER_TAB_PRIVS_MADE 来查看对象权限的授权信息。

【实例 24-27】查看 SCOTT 用户中表对象的授权信息。

```
SQL> col table_name for a10
SQL> col grantor for a15
SQL> col privilege for a10
SQL> select *
  2* from user_tab_privs_made

GRANTEE    TABLE_NAME GRANTOR        PRIVILEGE  GRA
------------------ -------------------- --------------------------- ---
LARRY       EMP        SCOTT           UPDATE     YES
```

从输出可以清楚地看出 GRANTOR 是 SCOTT 用户，而 GRANTEE 是 LARRY 用户，SCOTT 将拥有的表 EMP 的 UPDATE 对象权限赋予了 LARRY。

下面将 SCOTT 用户的某个表的某些列的对象权限赋予用户 sophie，如实例 24-28 所示。

【实例 24-28】把对表 DEPT 的列的 UPDATE 对象权限赋予用户 sophie。

```
SQL> grant update(dname,loc) on dept to sophie;

授权成功。
```

Oracle 提供了一个数据字典 USER_COL_PRIVS_MADE 用于记录用户的列对象权限的赋予情况，如实例 24-29 所示。

【实例 24-29】使用数据字典 USER_COL_PRIVS_MADE 查看相关列的权限赋予信息。

```
SQL> col column_name for a10
SQL> select *
  2  from user_col_privs_made;

GRANTEE    TABLE_NAME COLUMN_NAM GRANTOR    PRIVILEGE  GRA
------------------- ------------------- --------------------- -------
SOPHIE     DEPT       DNAME      SCOTT          UPDATE     NO
SOPHIE     DEPT       LOC         SCOTT          UPDATE     NO
```

从上述输出可以看出，当前用户 SCOTT 的表中列的权限赋予信息，其中表为 DEPT，而与其相关的列为 DNAME 和 LOC，将两列的 UPDATE 权限赋予用户 SOPHIE。

在本节中，使用 SCOTT 用户将表 EMP 的 UPDATE 权限赋予了用户 LARRY，且用户 LARRY 可以将该权限继续赋予其他用户，同时又将 DEPT 表的列 DNAME 和 LOC 赋予了用户 SOPHIE。

24.2.2 回收对象权限

出于安全的考虑，如果一个用户不需要某种对象权限可以使用 REVOKE 指令回收用户的对象权限。回收对象权限的语法格式如下：

```
REVOKE { object_privilege
         [, object privilege ] ……
         | ALL [ PRIVILEGE ] }
ON   [schema.] object
FROM   {user | role | PUBLIC}
         [, { user | role | PUBLIC } ] ……
         [ CASCADE CONSTRAINTS ]
```

回收用户 LARRY 和 SOPHIE 的对象权限如实例 24-30 所示。

【实例 24-30】回收用户 LARRY 对 EMP 表的 UPDATE 对象权限。

```
SQL> conn scott/tiger@bjyzz
已连接。
SQL> revoke update on emp
  2  from larry ;

撤销成功。
```

注意，上例中我们使用 SCOTT 用户登录数据库，然后使用数据字典 USER_TAB_PRIVS_MADE 来验证是否成功回收权限。

【实例 24-31】验证是否成功回收赋予用户 LARRY 的对象权限。

```
SQL> select *
  2  from user_tab_privs_made
  3  where grantee ='LARRY';

未选定行
```

显然，数据字典 USER_TAB_PRIVS_MADE 中没有记录用户 SCOTT 的表对象权限的授权信息。说明成功回收赋予用户 LARRY 的对象权限。

【实例 24-32】回收用户 SOPHIE 对 DEPT 表的列操作的权限。

```
SQL> revoke all on dept
  2  from sophie;

撤销成功。
```

在回收对象权限时，只能从整个表而不能按列回收，所以虽然按照列赋予用户 SOPHIE 的对象权限，但是不能按列权限回收，以下是一个错误示例。

```
SQL> revoke update(dname,loc) on dept
  2  from sophie;
revoke update(dname,loc) on dept
              *
ERROR 位于第 1 行:
ORA-01750: UPDATE/REFERENCES 仅可以从整个表而不能按列 REVOKE
```

在实例 24-32 中，成功回收了用户 SOPHIE 对表 DEPT 的列 DNAME 和 LOC 的 UPDATE 权限。下面通过实例 24-33 验证是否成功回收权限。

【实例 24-33】验证是否成功回收拥护 SOPHIE 的对象权限。

```
SQL> select *
  2  from user_col_privs_made;

未选定行
```

从输出结果可以看出，在数据字典 USER_COL_PRIVS_MADE 中没有记录用户 SOPHIE 的对象操作权限。



对象权限的回收是级联的，如用户 SCOTT 授予用户 LARRY 对象权限 A 且具有继续授权的能力，用户 LARRY 继续将对象权限 A 赋予用户 SOPHIE 且具有继续授权能力，用户 SOPHIE 可以继续授权，如果此时用户 SCOTT 回收用户 LARRY 的对象权限 A 则用户 SOPHIE 不再具有继续向其他用户授予对象权限 A 的能力。

24.3　本章小结

本章主要讲解了数据库系统权限和对象权限管理方面的知识，系统权限是与数据库系统相关的权限，如创建数据库、创建表空间等，它是 Oracle 数据库中具有最高权限的用户，可以实现对数据维护的任何工作。读者需要掌握授予和回收系统权限的方法，对象权限是指对数据库对象如表、视图、序列号和存储过程等操作的权利，这些权利包括 ALTER、UPDATE、DELETE 和 INSERT 等，在维护数据库对象时需要读者很好地掌握这些对象权限对应的数据库对象类型，并掌握如何授予和回收对象权限。

第 25 章

EXP/IMP及数据泵的备份与恢复

数据库备份是 DBA 的一项重要日常任务。没有备份就没有恢复，所以 DBA 需要选择良好的备份方案、合适的备份工具，以及相应的恢复方案。EXP/IMP 是 Oracle 比较传统地数据库逻辑备份工具，用于实现全库或表空间的逻辑备份，但是它不支持用户的交互模式，即在备份过程中无法控制或切换备份进程，而 Oracle 的数据泵技术可以很好地实现用户交互、支持网络操作以及重启失败的备份作业。

25.1　什么是备份

或许读者经常听到以下的说法，如逻辑备份和物理备份、一致备份和非一致备份、脱机备份和联机备份、备份粒度等。读者很好地理解并把握这些概念对于完成备份和恢复很重要。

1. 逻辑备份和物理备份

逻辑备份导出数据库的结构以及数据，这些结构包括表的定义、触发器、存储过程等数据库对象，当使用数据泵技术或 EXP/IMP 技术时实现的是逻辑备份。物理备份是将数据库的数据文件、控制文件和归档日志文件的重要文件拷贝到操作系统的其他磁盘，此时的文件保持原文件类型。

2. 脱机备份和联机备份

脱机备份是指在数据库关闭的情况下实现数据备份，也称为冷备份。而联机备份是数据运行时进行的数据备份，也叫做热备份。采用联机备份还是脱机备份依赖于业务的需求，对于 7*24 小时运行的数据库显然不能使用脱机备份，但是联机备份相对复杂，必须考虑数据库的归档模式，以及设计合理的联机备份方案。

3. 一致备份和非一致备份

先解释一下“恢复”（RECOVER）的概念，因为数据文件和控制文件中的系统 SCN 不一致，恢复进程必须使用归档日志文件和联机重做日志文件的数据更新数据文件中的内容，也就是将重做日志文件中用户提交的数据重新写入数据文件。再解释“一致”的概念，Oracle 为每个事务设置了一个唯一的 SCN（系统更改号），当每次事务提交时都自动增加 SCN 号，这个号码永远是唯一的。当 DBWR 写进程运行时，将触发一个检验点事件，将数据库缓冲区中所有已经提交的数据写入磁

盘，并使得所有数据文件和控制文件中的 SCN 相一致。这里一致的概念就是所有数据文件和控制文件中的 SCN 相同。

而当 LGWR 将数据库缓冲区中变化的数据写入重做日志文件时，对于用户提交了数据的事务的 SCN 将记录在控制文件中，注意此时的数据文件中的 SCN 没有变化，这就叫做不一致状态。

所以一致备份和不一致备份的区别就是是否需要恢复。要实现一致备份可以关闭数据库使用脱机备份的方式，也可以使数据库处于 MOUNT 状态，使用 RMAN 工具实现。

在 7*24 小时运行的数据库中，不一致备份是唯一的选择，它并不代表这样的备份是不可靠的，只是在数据恢复时需要一个“一致”的过程，只要数据库处于归档模式，且重做日志归档文件没有损坏就可以使用不一致的备份实现数据库的完全恢复，不会造成数据的丢失。

25.2　EXP 指令

EXP 和 IMP 是 Oracle 比较“古老”的数据备份和恢复方式，使用 EXP 实用程序可以导出整个数据库、一个用户的所有对象、一个表空间或特定的表。使用 EXP 实用程序导出的数据必须使用 IMP 实用程序恢复备份的数据，本节将讲解 EXP 备份数据的方法，以及给出备份不同数据库对象的实例，如备份整个数据库、备份特定的表空间和部分表。

25.2.1　EXP 指令详解

Oracle 的 EXP 实用程序使用命令行方式备份数据，所以它支持一系列的操作指令，通过这些指令告诉数据库要备份的数据类型，以及登录数据库的用户名和密码，以及其他和导出数据相关的关键字。如果用户在 Windows 平台上，可以在 DOS 窗口中输入 EXP HELP = Y，然后回车，查看 EXP 的数据导出指令，如实例 25-1 所示。

【实例 25-1】EXP 实用程序的参数指令。

```
C:\Documents and Settings\Administrator>exp help=y

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

通过输入 EXP 命令和您的用户名/口令，导出
操作将提示您输入参数:

     例如: EXP SCOTT/TIGER

或者，您也可以通过输入跟有各种参数的 EXP 命令来控制导出
的运行方式。要指定参数，您可以使用关键字:

     格式:  EXP KEYWORD=value 或 KEYWORD=(value1,value2,...,valueN)
     例如: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
               或 TABLES=(T1:P1,T1:P2)，如果 T1 是分区表
```

```
USERID 必须是命令行中的第一个参数。

关键字    说明 (默认值)          关键字      说明 (默认值)
--------------------------------------------------------------------------
USERID   用户名/口令            FULL        导出整个文件 (N)
BUFFER   数据缓冲区大小         OWNER        所有者用户名列表
FILE     输出文件 (EXPDAT.DMP)  TABLES       表名列表
COMPRESS  导入到一个区 (Y)   RECORDLENGTH   IO 记录的长度
GRANTS    导出权限 (Y)           INCTYPE     增量导出类型
INDEXES   导出索引 (Y)          RECORD       跟踪增量导出 (Y)
DIRECT    直接路径 (N)          TRIGGERS     导出触发器 (Y)
LOG      屏幕输出的日志文件    STATISTICS    分析对象 (ESTIMATE)
ROWS      导出数据行 (Y)         PARFILE      参数文件名
CONSISTENT 交叉表的一致性 (N)   CONSTRAINTS  导出的约束条件 (Y)

OBJECT_CONSISTENT    只在对象导出期间设置为只读的事务处理 (N)
FEEDBACK            每 x 行显示进度 (0)
FILESIZE            每个转储文件的最大大小
FLASHBACK_SCN        用于将会话快照设置回以前状态的 SCN
FLASHBACK_TIME       用于获取最接近指定时间的 SCN 的时间
QUERY               用于导出表的子集的 select 子句
RESUMABLE            遇到与空格相关的错误时挂起 (N)
RESUMABLE_NAME       用于标识可恢复语句的文本字符串
RESUMABLE_TIMEOUT    RESUMABLE 的等待时间
TTS_FULL_CHECK       对 TTS 执行完整或部分相关性检查
TABLESPACES          要导出的表空间列表
TRANSPORT_TABLESPACE 导出可传输的表空间元数据 (N)
TEMPLATE            调用 iAS 模式导出的模板名

成功终止导出，没有出现警告。
```

上述输出是 Oracle 11g 数据库中的 EXP 参数指令，虽然在 Oracle 11g 中建议使用 Oracle 的数据泵导入、导出数据库技术，但是出于兼容性方面的考虑在 Oracle11g 版本中仍然支持 EXP 实用程序，但是它和 Oracle 9i 中的参数指令略有不同。下面解释部分的参数指令。

- USERID：该参数无默认值，说明登录数据库的用户名/密码。
- BUFFER：指定数据缓冲区的大小，该参数依赖于特定的操作系统，用户可以根据导出数据的性质，如 LOB 数据类型可以适当设置较大的 BUFFER 参数值。
- FILE：说明将备份的数据文件重新命名。该输出文件名默认为 EXPDAT.DMP 或者是 expdat.dmp。
- COMPRESS：该参数的默认值为 Y 或 N，使 Oracle 对输入文件进行配置，使得当引入并且重新创建对象时，对象初始化的大小为已经导出的对象大小。
- GRANTS：该参数值为 Y 或 N，用来控制授权的导出。
- INDEXES：该参数值为 Y 或 N，用来控制索引的导出。
- LOG：该参数没有默认值，说明在导出备份时是否需要创建一个备份日志，该日志用于记录整个备份过程。
- FULL：说明是否导出整个数据库的所有对象，该参数值为 N 或 Y，如果执行整个数据库

的导出，则连接的用户必须具有 DBA 权限或者具有 EXP_FULL_DATABASE 权限。

- OWNER：该参数没有默认值，OWNER 参数说明导出特定用户的数据库对象，可以有多个用户名，使用逗号隔开。
- TABLES：说明要导出的表的名称，如果该表是属于当前连接用户的，则直接输入表名，否则输入模式名.表名，此处可以使用多个表名，使用逗号隔开。
- TRIGGERS：该参数的默认值为 Y 或 N，说明是否导出该用户模式下的触发器对象。
- STATISTICS：该参数的值为 estimate、computer 或者 none，指出产生的对象统计量的方式，如果选择了 ESTIMATE、COMPUTER，则以该方式计算统计量，如果为 NONE，则不使用统计量。
- CONSTRAINTS：该参数的值为 Y 或 N，说明是否要导出约束。
- FEEDBACK：每隔一定的行数显示备份进展情况，该参数的值从 0 到任何有效的数字。
- FILESIZE：指出每个转储文件的最大值。
- TABLESPACES：说明要导出的表空间的名称，使得 EXP 程序可以从这些表空间中导出数据，该参数可以有多个表空间名，用逗号隔开。
- RESUMABLE：该参数的值为 Y 或 N，指出 EXP 是否使用 Oracle 的可恢复空间管理工具，使用该工具使得导出数据时，如发生与空间相关的错误应该中止导出。

当然，在使用 EXP 程序时也可以不使用任何参数，此时需要用户输入一些参数来导出要备份的数据，如用户名和密码、输入缓冲区大小、导出的文件名、导出表还是导出所有用户对象、是否导出权限等。

可以使用不带参数的 EXP 指令备份数据库，不过系统会提示输入一系列参数来完成备份。下面给出一个实例以更详细地理解这些参数。

【实例 25-2】使用 EXP 实用程序备份 SCOTT 用户的所有数据库对象。

```
C:\Documents and Settings\Administrator>exp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


用户名: scott@orcl
口令:

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
输入数组提取缓冲区大小: 4096 >

 导出文件: EXPDAT.DMP >

(2)U(用户), 或 (3)T(表): (2)U >

导出权限 (yes/no): yes >
```

```
导出表数据 (yes/no): yes >

压缩区 (yes/no): yes >

已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集
. 正在导出 pre-schema 过程对象和操作
. 正在导出用户 SCOTT 的外部函数库名
. 导出 PUBLIC 类型同义词
. 正在导出专用类型同义词
. 正在导出用户 SCOTT 的对象类型定义
即将导出 SCOTT 的对象...
. 正在导出数据库链接
. 正在导出序号
. 正在导出簇定义
. 即将导出 SCOTT 的表通过常规路径...
. . 正在导出表          BACKUP_DELETE_EMP_TABLE 导出了          0 行
. . 正在导出表                        BONUS 导出了            0 行
. . 正在导出表          CREATE$JAVA$LOB$TABLE 导出了          1 行
. . 正在导出表                         DEPT 导出了            4 行
. . 正在导出表                    DEPT_TEST 导出了            5 行
. . 正在导出表                          EMP 导出了           28 行
. . 正在导出表                    EMP_TEST2 导出了            0 行
. . 正在导出表                    EMP_TEST3 导出了          2 行
. . 正在导出表                        ROOMS 导出了         4 行
. . 正在导出表                     SALGRADE 导出了          5 行
. . 正在导出表            USER_MODIFY_TABLE 导出了         20 行
. 正在导出同义词
. 正在导出视图
. 正在导出存储过程
. 正在导出运算符
. 正在导出引用完整性约束条件
. 正在导出触发器
. 正在导出索引类型
. 正在导出位图，功能性索引和可扩展索引
. 正在导出后期表活动
. 正在导出实体化视图
. 正在导出快照日志
. 正在导出作业队列
. 正在导出刷新组和子组
. 正在导出维
. 正在导出 post-schema 过程对象和操作
. 正在导出统计信息
成功终止导出，没有出现警告。
```

EXP 指令是在操作系统下运行的，导出的文件保存在操作 EXP 指令的当前操作系统目录下。一旦输入 EXP 指令然后回车，就提示输入用户名和密码，如果二者都正确，则提示连接到数据库，最后是一系列参数选择，下面依次解释这些参数。

- 输入数组提取缓冲区大小（4096＞）：缓冲区大小在 WINDOWS 上默认为 4096 字节，如果用户包含大对象(LOB)，则设置该参数至少为 2M。

- 导出文件（EXPDAT.DMP >）：导出的备份文件默认文件名为 EXPDAT.DMP，也可以根据需要自己的需要命名，在“>”后输入自定义的导出备份文件名。
- (2)U(用户)或 (3)T(表)（(2)U >）：导出当前用户的所有数据库对象，还是导出当前用户的表，冒号后是默认值，默认是导出用户的数据库对象。如果选择导出表，则提示如下信息，使用这种方式导出用户表时，每次只能导出当前用户的一个表，所以如果有多个表要导出，则需要多个操作步骤。读者可以根据自己的需求选择是否使用这种方式。

```
(2)U(用户), 或 (3)T(表): (2)U > t

导出表数据 (yes/no): yes >

压缩区 (yes/no): yes >

已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集

即将导出指定的表通过常规路径...
要导出的表 (T) 或分区 (T: P): (按 RETURN 退出) > dept

. . 正在导出表                          DEPT 导出了           4 行
```

- 导出权限（(yes/no): yes >）：选择是否导出权限，即是否导出对表/视图/序列/角色的授权。
- 导出表数据（(yes/no): yes >）：是否导出表数据，Oracle 提供了这样一种方式既可以导出表的结构，也可以导出数据，或者导出表结构和表数据，如果不导出表数据则只导出表的结构，也就是导出表的定义，模式选择导出表数据。
- 压缩区（(yes/no): yes >）：选择是否使用压缩区。

25.2.2　通过 EXP 指令导出整个数据库

导出整个数据库比较简单，只需要 FULL 和 FILE 两个参数。FULL 表示导出整个文件，包括所有的数据库对象，而 FILE 表示备份后的数据库文件名，可以自定义，如实例 25-3 所示。使用 SYSTEM 用户登录数据库，导出整个数据库。

【实例 25-3】EXP 指令备份整个数据库。

```
D:\>exp system/oracle@orcl full=y file = backup wholedatabase.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0  - Production
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集

即将导出整个数据库...
. 正在导出表空间定义
```

```
. 正在导出概要文件
. 正在导出用户定义
. 正在导出角色
. 正在导出资源成本
. 正在导出回退段定义
. 正在导出数据库链接
. 正在导出序号
. 正在导出目录别名
. 正在导出上下文名称空间
. 正在导出外部函数库名
. 导出 PUBLIC 类型同义词
. 正在导出专用类型同义词
. 正在导出对象类型定义
. 正在导出系统过程对象和操作
. 正在导出 pre-schema 过程对象和操作
. 正在导出簇定义
. 即将导出 SYSTEM 的表通过常规路径...
. . 正在导出表                    DEF$_AQCALL 导出了          0 行
. . 正在导出表                   DEF$_AQERROR 导出了          0 行
。。。。。。。。。
. . 正在导出分区                    COSTS_1995 导出了          0 行
. . 正在导出分区                    COSTS_1996 导出了          0 行
。。。。。。。。。
. 即将导出 PM 的表通过常规路径...
. . 正在导出表                   ONLINE_MEDIA 导出了          9 行
. . 正在导出表                    PRINT_MEDIA 导出了          4 行
. . 正在导出表              TEXTDOCS_NESTEDTAB 导出了         12 行
. 即将导出 BI 的表通过常规路径...
. 即将导出 CAT 的表通过常规路径...
. 正在导出同义词
. 正在导出视图
. 正在导出引用完整性约束条件
. 正在导出存储过程
. 正在导出运算符
. 正在导出索引类型
. 正在导出位图，功能性索引和可扩展索引
. 正在导出后期表活动
. 正在导出触发器
. 正在导出实体化视图
. 正在导出快照日志
. 正在导出作业队列
. 正在导出刷新组和子组
. 正在导出维
. 正在导出 post-schema 过程对象和操作
. 正在导出用户历史记录表
. 正在导出默认值和系统审计选项
. 正在导出统计信息
导出成功终止，但出现警告。
```

输出显示已导出成功，备份的全库文件名为 backup_wholedatabase_090825.dmp，从备份输出

信息可以看出开始导出了数据库对象的定义，如表空间定义、概要文件定义以及用户定义，然后导出数据库中的表、分区，最后导出数据库中各类对象的定义。总之，通过全库导出，导出了数据库中对象的定义，以及各种表结构以及表中的数据，这种备份也可以称为逻辑备份。

如果只需要导出数据库的结构而不需要表中的数据，则可以使用如实例 25-4 所示的方法。

【实例 25-4】使用 EXP 指令导出不包含数据的全库备份。

```
D:\>exp system/oracle@orcl full =y rows =n file =myfull withoutdata.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Produ
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集
注: 将不导出表数据 (行)

即将导出整个数据库...
. 正在导出表空间定义
……
```

为了节约篇幅，只给出部分输出结果，省略的剩余部分和上例中的相应部分相同，唯一的区别是只导出对象定义，而不导出数据。

参数 ROWS=no，告诉 EXP 程序要导出的数据库不包含数据，而只导出整个数据库中数据库对象的定义。

25.2.3　通过 EXP 指令导出特定的表

导出特定用户的特定的表需要 TABLES 参数，该参数后可以有几个表名，如果需要导出的表不是当前用户的表，则需要使用 schema_name.table_name 的形式，告诉 EXP 程序该表属于哪个用户。使用 FILE 参数自定义备份文件名。下面给出一个实例，使用 SYSTEM 用户登录，导出 SCOTT 用户的两个表 DEPT 和 EMP。

【实例 25-5】使用 EXP 指令导出特定的表。

```
D:\>exp system/oracle@orcl tables = scott.dept,scott.emp file = tables.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集
```

```
即将导出指定的表通过常规路径...
当前的用户已更改为 SCOTT
. . 正在导出表                          DEPT 导出了          4 行
. . 正在导出表                           EMP 导出了         28 行
成功终止导出, 没有出现警告。
```

在上述输出中显示导出成功，导出的文件名为 tables.dmp，因为用户 SYSTEM 具有对用户 SCOTT 中表的访问权，所以该用户可以导出 SCOTT 用户的表，如果使用 SCOTT 用户登录数据库，在导出自己的表时可以直接写出表名，而不必使用模式名，如实例 25-6 所示。

【实例 25-6】使用 EXP 导出当前用户的表。

```
D:\>exp scott/oracle@orcl tables = dept,emp file = backup scott tables.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集

即将导出指定的表通过常规路径...
. . 正在导出表                          DEPT 导出了          4 行
. . 正在导出表                           EMP 导出了         28 行
成功终止导出, 没有出现警告。
```

上例中使用 SCOTT 用户登录，而导出的表也是该用户拥有的表，所以不需要指定模式名。

25.2.4 通过 EXP 指令导出指定的用户

如果不需要导出整个数据库，而是导出当前数据库中某个用户的所有数据库对象，作为该用户的数据库备份，则使用 OWNER 参数指定，如实例 25-7 所示。

【实例 25-7】使用 EXP 指令导出 SCOTT 用户的所有数据库对象。

```
D:\>exp system/oracle@orcl owner = scott file = userbackup scott.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集

即将导出指定的用户...
. 正在导出 pre-schema 过程对象和操作
. 正在导出用户 SCOTT 的外部函数库名
```

```
. 导出 PUBLIC 类型同义词
. 正在导出专用类型同义词
. 正在导出用户 SCOTT 的对象类型定义
即将导出 SCOTT 的对象...
. 正在导出数据库链接
. 正在导出序号
. 正在导出簇定义
. 即将导出 SCOTT 的表通过常规路径...
. . 正在导出表          BACKUP_DELETE_EMP_TABLE 导出了         0 行
. . 正在导出表                            BONUS 导出了         0 行
. . 正在导出表          CREATE$JAVA$LOB$TABLE 导出了           1 行
. . 正在导出表                             DEPT 导出了         4 行
. . 正在导出表                        DEPT_TEST 导出了         5 行
. . 正在导出表                              EMP 导出了        28 行
. . 正在导出表                        EMP_TEST2 导出了         0 行
. . 正在导出表                        EMP_TEST3 导出了         2 行
. . 正在导出表                            ROOMS 导出了         4 行
. . 正在导出表                         SALGRADE 导出了         5 行
. . 正在导出表                USER_MODIFY_TABLE 导出了        20 行
. 正在导出同义词
. 正在导出视图
. 正在导出存储过程
. 正在导出运算符
. 正在导出引用完整性约束条件
. 正在导出触发器
. 正在导出索引类型
. 正在导出位图，功能性索引和可扩展索引
. 正在导出后期表活动
. 正在导出实体化视图
. 正在导出快照日志
. 正在导出作业队列
. 正在导出刷新组和子组
. 正在导出维
. 正在导出 post-schema 过程对象和操作
. 正在导出统计信息
成功终止导出，没有出现警告。
```

导出特定用户的所有数据库对象，其实是导出该用户所拥有的数据库对象的定义，以及对象所拥有的数据。从导出输出中可以清楚地看到导出了这些对象的定义，以及表中的数据。

上例中我们使用 SYSTEM 用户登录数据库，该用户具有 DBA 权限，所以它拥有访问 SCOTT 用户的数据库对象权限，所以可以成功导出 SCOTT 用户的所有数据库对象。其实，也完全可以使用 SCOTT 用户登录数据库，使用 EXP 指令导出它自己拥有的所有数据库对象，如实例 25-8 所示。

【实例 25-8】使用 EXP 指令导出当前用户的数据库对象。

```
D:\>exp scott/oracle@orcl owner = scott file = bakup_user_scott.dmp
```

此处，我们只给出指令，而不显示具体结果，其备份过程与例 25-7 的输出一样。

25.2.5 通过 EXP 指令导出特定的表空间

在数据库维护时，可能只需要重点维护一个表空间，如 USERS 表空间，该表空间用于放置用户数据。此时可使用 EXP 指令的 TABLESPACES 参数来指定当前用户所拥有的特定表空间，如实例 25-9 所示。

【实例 25-9】使用 EXP 指令导出特定表空间。

```
D:\>exp system/oracle@orcl tablespaces =users file = backup_tablespaces_users.dmp

Export: Release 11.1.0.6.0 - Production on 星期五 10 月 16 15:40:20 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已导出 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集

即将导出所选表空间...
对于表空间 USERS...
. 正在导出簇定义
. 正在导出表定义
. . 正在导出表          BACKUP_DELETE_EMP_TABLE 导出了          0 行
. . 正在导出表                           BONUS 导出了          0 行
. . 正在导出表            CREATE$JAVA$LOB$TABLE 导出了          1 行
. . 正在导出表                            DEPT 导出了          4 行
. . 正在导出表                       DEPT_TEST 导出了          5 行
. . 正在导出表                             EMP 导出了         28 行
. . 正在导出表                       EMP_TEST2 导出了          0 行
. . 正在导出表                       EMP_TEST3 导出了          2 行
. . 正在导出表                           ROOMS 导出了          4 行
. . 正在导出表                        SALGRADE 导出了          5 行
. . 正在导出表               USER_MODIFY_TABLE 导出了         20 行
. . 正在导出表                  CATEGORIES_TAB 导出了         22 行
. . 正在导出表         PRODUCT_REF_LIST_NESTEDTAB 导出了        288 行
. . 正在导出表      SUBCATEGORY_REF_LIST_NESTEDTAB 导出了         21 行
EXP-00079: 表 "PURCHASEORDER" 中的数据是被保护的。常规路径只能导出部分表。
. . 正在导出表                   PURCHASEORDER 导出了        132 行
. 正在导出引用完整性约束条件
. 正在导出触发器
导出成功终止，但出现警告。
```

EXP 将针对特定的表空间实施导出操作，导出该表空间中所有表的定义和表中的数据。如果不需要导出表中的数据，只需要备份表空间中对象的定义，可以使用 ROWS 参数，设置 rows = no 即可。

25.3　IMP 指令

IMP 指令用于导入通过 EXP 指令导出的备份数据，IMP 可以导入一个完整的数据库、一个指定的用户的所有数据库对象、一个特定的表空间以及一个特定的表。在说明如何使用 IMP 指令导入数据前，先给出 IMP 指令的参数指令，以实现数据库的恢复。

25.3.1　IMP 指令详解

IMP 指令同样是运行在操作系统上的，通过输入 IMP 指令和各种参数来控制数据导入的运行方式，在 Windows 的 DOS 窗口中输入 emp help =y ，然后回车，显示如实例 25-10 所示的 IMP 参数信息。

【实例 25-10】IMP 实用程序的参数指令。

```
C:\Documents and Settings\Administrator>imp help=y

Import: Release 11.1.0.6.0 - Production on 星期二 8月 25 07:55:12 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

通过输入 IMP 命令和您的用户名/口令, 导入
操作将提示您输入参数:

    例如: IMP SCOTT/TIGER

或者, 可以通过输入 IMP 命令和各种参数来控制导入
的运行方式。要指定参数, 您可以使用关键字:

    格式:  IMP KEYWORD=value 或 KEYWORD=(value1,value2,...,valueN)
    例如: IMP SCOTT/TIGER IGNORE=Y TABLES=(EMP,DEPT) FULL=N
            或 TABLES=(T1:P1,T1:P2), 如果 T1 是分区表

USERID 必须是命令行中的第一个参数。

关键字   说明 (默认值)         关键字      说明 (默认值)
--------------------------------------------------------------------------
USERID   用户名/口令            FULL        导入整个文件 (N)
BUFFER   数据缓冲区大小         FROMUSER    所有者用户名列表
FILE     输入文件 (EXPDAT.DMP)  TOUSER      用户名列表
SHOW     只列出文件内容 (N)     TABLES      表名列表
IGNORE   忽略创建错误 (N)      RECORDLENGTH IO 记录的长度
GRANTS   导入权限 (Y)           INCTYPE     增量导入类型
INDEXES  导入索引 (Y)           COMMIT      提交数组插入 (N)
ROWS     导入数据行 (Y)         PARFILE     参数文件名
LOG      屏幕输出的日志文件     CONSTRAINTS 导入限制 (Y)
DESTROY                  覆盖表空间数据文件 (N)
INDEXFILE                将表/索引信息写入指定的文件
```

```
SKIP_UNUSABLE_INDEXES  跳过不可用索引的维护 (N)
FEEDBACK              每 x 行显示进度 (0)
TOID_NOVALIDATE        跳过指定类型 ID 的验证
FILESIZE              每个转储文件的最大大小
STATISTICS            始终导入预计算的统计信息
RESUMABLE             在遇到有关空间的错误时挂起 (N)
RESUMABLE_NAME         用来标识可恢复语句的文本字符串
RESUMABLE_TIMEOUT      RESUMABLE 的等待时间
COMPILE               编译过程, 程序包和函数 (Y)
STREAMS_CONFIGURATION  导入流的一般元数据 (Y)
STREAMS_INSTANTIATION  导入流实例化元数据 (N)

下列关键字仅用于可传输的表空间
TRANSPORT_TABLESPACE 导入可传输的表空间元数据 (N)
TABLESPACES 将要传输到数据库的表空间
DATAFILES 将要传输到数据库的数据文件
TTS_OWNERS 拥有可传输表空间集中数据的用户

成功终止导入，没有出现警告。
```

下面详细解释部分参数的含义。

- USERID：说明登录数据库的用户名和密码。
- BUFFER：说明输入数据缓冲区的大小。
- FILE：说明通过 EXP 程序创建的备份文件名，有时需要绝对路径。
- SHOW：说明是否使导入过程只显示备份文件。
- IGNORE：说明是否忽略在输入过程中备份文件中的错误。
- GRANTS：说明是否导入备份文件中的授权。
- INDEXES：说明是否导入索引。
- ROWS：是否导入数据行，该参数的值为 y 或者 n，如果 ROWS = n，则不输入数据。
- LOG：是否将导入过程记录到日志文件。
- FULL：对整个备份文件完全导入。
- FROMUSER：允许将一个备份文件中的对象从一个用户复制到另一个用户，该参数说明导入数据的源用户名。
- TOUSER：允许将一个备份文件中的对象从一个用户复制到另一个用户，该参数说明导入数据的目的户名。
- TABLES：说明要导入的表名列表，如果是多个表，则使用逗号分开。
- SKIP_UNUSABLE_INDEXES：该参数值为 y 或 n，说明是否需要重建已经设置为 unusable 状态的索引。
- STATISTICS：说明在导入对象后对统计量如何处理，该参数值为 always、none、safe 或者 recalculate。其中 always 表示从备份文件中导入统计量，none 表示不执行任何动作；safe 表示如果使用了可靠的优化器，则允许导入优化器统计量，recalculate 表示需要重新计算统计量，而不是从备份文件中导入统计量。
- CONSTRAINTS：说明是否导入备份数据中的约束。

● COMPILE：该参数说明是否引入进程来编译过程、包和函数。

25.3.2　通过 IMP 指令恢复整个数据库

IMP 使用程序将备份的整个数据库导入当前的数据库中。在导入过程中包括一系列创建数据库对象的过程，如创建表空间、表的索引、序列号以及用户授权等。如果要创建的数据库对象，如某个表已经存在，则该创建语句会失败。此时需要使用 ignore 参数，该参数使得 IMP 程序忽略这些错误。下面是使用了 ignore 参数的全库恢复的实例。

【实例 25-11】使用 IMP 指令导入整个数据库。

```
D:\>imp system/oracle@orcl full =y file=backup_wholedatabase.dmp ignore =y
```

上例中参数 file 后的文件名说明该文件位于当前使用 IMP 指令的目录下，参数 ignore=y 使得在创建数据库对象时不会造成由于对象已经存在导致输入操作产生错误。

如果用户的数据量比较大，这个过程将很“漫长”，使用这种方法恢复整个数据库其实就是重新创建数据库中所有对象的过程，所以它是一种逻辑数据库恢复的方法。

25.3.3　通过 IMP 指令恢复特定的表

在第 25.3.4 小节中，使用 EXP 指令导出用户 SCOTT 的两个表 DEPT 和 EMP，如果由于某种原因删除了表 EMP，然后需要恢复 EMP 表（当然可以通过 Oracle 的脚本文件重建 SCOTT 用户的所有数据库对象，这里模拟恢复特定的表），则可以使用 EXP 程序备份的导出文件 backup_scott_tables.dmp，如实例 25-12 所示。

【实例 25-12】删除用户 SCOTT 的 EMP 表。

```
SQL> drop table emp;

表已删除。
```

然后查询该表是否存在，如实例 25-13 所示。

【实例 25-13】验证表 EMP 是否存在。

```
SQL> desc emp;
ERROR:
ORA-04043: 对象 emp 不存在
```

显然，输出结果说明已经成功删除了表 EMP，下面使用 IMP 程序来恢复这个表，如实例 25-14 所示。

【实例 25-14】使用 IMP 指令恢复特定的表。

```
D:\>imp scott/oracle@orcl tables =emp file =backup scott tables.dmp

Import: Release 11.1.0.6.0 - Production on 星期二 8月 25 23:27:21 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
```

```
With the Partitioning, OLAP and Data Mining options

经由常规路径由 EXPORT:V11.1.0.6.0 创建的导出文件
已经完成 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集中的导入
. 正在将 SCOTT 的对象导入到 SCOTT
. 正在将 SCOTT 的对象导入到 SCOTT
. . 正在导入表                    "EMP"导入了         28 行
成功终止导入，没有出现警告。
```

显然，成功从备份文件 backup_scott_tables.dmp 中恢复了用户 SCOTT 的表 EMP。

下面再通过 DESC 指令查看表 EMP 的定义是否存在，如实例 25-15 所示。

【实例 25-15】查看恢复后表 EMP 是否存在。

```
SQL> desc emp;
 名称                                     是否为空? 类型
 ----------------------------------------- -------- --------------
 EMPNO                                              NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
 DEPTNO                                             NUMBER(2)
```

25.3.4 通过 IMP 指令恢复指定的用户

在一个数据库中可以创建多个用户，每个用户有自己的数据库对象，如表、表空间、表索引、序列号、约束等。使用 EXP 程序导出指定的用户其实就是导出该用户所拥有的所有数据库对象。如果某用户的数据库对象需要恢复，则可使用 IMP 指令恢复用户的整个数据库对象，部分表或者将该用户的表导入其他用户。

【实例 25-16】导入 SCOTT 用户的全部数据库对象。

```
D:\>imp system/oracle@orcl full =y file =userbackup_scott.dmp

Import: Release 11.1.0.6.0 - Production on 星期三 8 月 26 00:04:41 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Productio
With the Partitioning, OLAP and Data Mining options

经由常规路径由 EXPORT:V11.1.0.6.0 创建的导出文件
已经完成 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集中的导入
. 正在将 SYSTEM 的对象导入到 SYSTEM
. 正在将 SCOTT 的对象导入到 SCOTT
. . 正在导入表       "BACKUP_DELETE_EMP_TABLE"导入了          0 行
```

```
. . 正在导入表                    "BONUS"导入了            0 行
. . 正在导入表       "CREATE$JAVA$LOB$TABLE"导入了            1 行
. . 正在导入表                     "DEPT"导入了            4 行
. . 正在导入表                "DEPT_TEST"导入了            5 行
. . 正在导入表                      "EMP"导入了           28 行
. . 正在导入表                "EMP_TEST2"导入了            0 行
. . 正在导入表                "EMP_TEST3"导入了            2 行
. . 正在导入表                    "ROOMS"导入了            4 行
. . 正在导入表                 "SALGRADE"导入了            5 行
. . 正在导入表        "USER_MODIFY_TABLE"导入了           20 行
即将启用约束条件...
成功终止导入，没有出现警告。
```

在上例中，使用了第 25.3.5 小节中导出的用户 SCOTT 的用户对象数据导入整个数据库对象。因为该文件当时是具有 DBA 权限的用户导出的，所以导入时也必须使用 DBA 权限的用户导入。

此例中必须指定 FULL 参数，如果需要导入特定的表数据可以使用 TABLES 参数，这样可以把特定的表导入到新的用户，如实例 25-17 所示。

【实例 25-17】将特定的表导入指定用户。

```
D:\>imp system/oracle@orcl tables =emp fromuser=scott touser=system file
=userbackup_scott.dmp

Import: Release 11.1.0.6.0 - Production on 星期三 8 月 26 00:20:46 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options

经由常规路径由 EXPORT:V11.1.0.6.0 创建的导出文件
已经完成 ZHS16GBK 字符集和 AL16UTF16 NCHAR 字符集中的导入
. 正在将 SCOTT 的对象导入到 SYSTEM
. . 正在导入表                      "EMP"导入了       28 行
"CREATE TRIGGER "SYSTEM".user change empdata"
 "before update or insert on emp"
 "for each row"
 "begin"
 " if inserting then"
 "insert into user_modify_table"
 " values (user,sysdate,'inserting');"
 "end if;"
 "if updating then"
 "insert into user_modify_table"
 " values (user,sysdate,'updating');"
 "end if;"
 "end user_change_empdata;"
 "CREATE TRIGGER "SYSTEM".when_backup_emp_trigger"
 "before delete on emp"
 "for each row"
```

```
"when (old.job IN 'MANAGER,PRESIDENT')"
"begin"
"insert into backup_delete_emp_table"
"values(:old.empno,:old.ename,:old.job,:old.mgr,:old.hiredate,"
"      :old.sal,:old.comm,:old.deptno);"
"end when backup emp trigger;"
成功终止导入，没有出现警告。
```

在上例中，我们向用户 SYSTEM 导入了 SCOTT 用户的表 EMP，当导入 EMP 表时其实就是在新用户 SYSTEM 的用户表空间中创建一个表，并填充数据，最后创建基于该表的触发器。如果读者自己创建了一个用户，一定要注意该用户必须具有对存放新表的表空间的访问权，否则提示对表空间无权限，无法恢复数据库对象到指定的用户。

25.4 数据泵

上两节讲解了如何使用 EXP 和 IMP 实用程序备份、恢复数据库，但是在 Oracle 11g 及以上版本中建议使用数据泵来代替 EXP 和 IMP 实用程序，本节将讲解如何使用数据泵技术特性，以及如何使用 Oracle 11g 的数据泵技术导入和导出数据。

在 Oracle 11g 中使用数据泵技术实现数据的导出和导入数据库。数据泵技术提供了许多新的特性，如可以中断导出、导入作业，再恢复作业的执行，从一个会话中监控数据泵取作业，在作业执行过程中修改作业属性，以及重启一个失败的数据泵取作业等。

25.4.1 数据泵概念详解

1. 数据泵导入导出技术的结构

在数据泵导入导出技术中，涉及导入实用程序 expdp 和导出实用程序 impdp，当启动数据泵导入或导出程序时，在数据库服务器端启动相应的服务器进程，完成数据的导入及导出任务，所以也称数据泵技术是基于 Oracle 数据库服务器的，导入及导出的数据文件也保存在数据库服务器端。为了说明数据泵导入及导出的结构，给出图示说明，如图 25-1 所示。

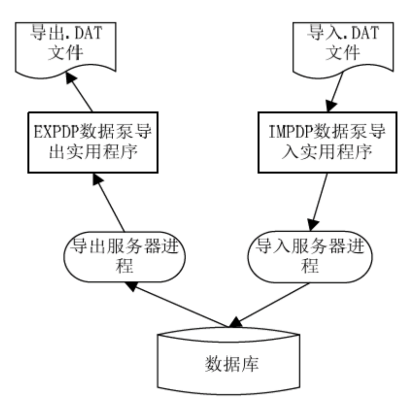


图 25-1　数据泵导入导出结构图

传统的 EXP 实用程序是一个普通的本地用户进程，它将备份的数据写入本地磁盘空间。EXP 实用程序是普通会话的一部分，它从服务器进程中获得要备份的数据。而数据泵取，即 EXPDP 程序启动数据库服务器端的服务器进程，服务器进程完成数据的备份并将备份文件写入数据库服务器端的计算机磁盘空间，文件格式为 filename.dat。导出的备份文件在导入时只能通过数据泵的导入实用程序 IMPDP 完成，将数据导入到运行在其他平台上的数据库中。


说　明

在使用数据泵取技术导出备份数据时，只能将备份的数据写入磁盘文件，而无法写入磁带设备，如果需要，必须通过第三方的工具实现。

2. 数据泵导入导出与 EXP/IMP 技术的区别

Oracle 数据泵技术是对传统的 EXP 和 IMP 实用程序的扩展，使得在数据库服务器端快速地移动数据。这里给出二者的主要区别，使得读者在使用时根据需要有所取舍。

- 数据泵技术比传统的 EXP/IMP 技术更快速地移动大量数据，因为数据泵技术采用并行流技术实现快速的并行处理。
- 数据泵技术基于数据库服务器，在启动数据泵导入导出实用程序时在数据库服务器端产生服务器进程负责备份或导入数据，并且将备份的数据备份在数据库服务器端。而且服务器进程与 EXPDP 客户机建立的会话无关。
- 传统的 EXP/IMP 是类似于普通的用户进程，执行如 SELECT、INSERT、CREATE 等 SQL 语句。而数据泵技术类似于启动作业的控制进程，不但启动客户端进程建立会话，还控制整个导入或导出过程，如重启作业。
- 使用传统的 EXP/IMP 实用程序导出的数据格式与数据泵技术导出的数据格式不兼容。
- 数据泵技术与传统的导入导出实用程序不同，它使用目录和目录对象存储数据泵导出文件，使用数据泵导出数据前必须先创建目录对象，否则无法使用数据泵导入和导出作业。

3. 数据泵导入导出技术的优点

在 Oracle 11g 中，Oracle 更新了 EXP/IMP 实用程序，对其进行了功能扩展，实现了这些实用程序的更新版本，即数据泵导入和导出技术。数据泵技术同样由两部分组成：数据泵导出程序（EMPDP）和数据泵导入程序（IMPDP），前者从数据库备份数据，后者从备份文件恢复到数据库中，通过调用 EXPDP 和 IMPDP 启动这两个数据泵导出和导入实用程序。数据泵技术基于数据库服务器，而传统的 EXP/IMP 实用程序基于客户机运行。

使用数据泵技术的优点如下所示。

- 导入导出速度更快：因为在数据泵导入导出作业中可以启动多个线程，所以可以并行的实现作业，对于移动大数据量时，性能显著提高。
- 重启失败的作业：这个功能是传统的 EXP/IMP 程序无法实现的，不论是数据泵导入导出作业停止还是失败，都可以很容易地重启作业。同时也支持手动停止或重启作业。
- 实时交互能力：在一个运行的数据泵作业中，可以从其他屏幕或控制终端与当前数据泵作业交互，以监控作业的执行以及更改作业的某些参数。

- 独立于客户机：因为数据泵技术是基于数据库服务器的，它是数据库服务器的一部分，一旦启动数据泵作业，则与客户机无关。
- 支持网络操作：支持在两个联网的数据库服务器之间导入和导出数据文件，也支持直接的将数据从一个数据库导入到另一个数据库，而不需要备份文件。网络操作的方式基于数据库连接，在数据库间直接移动数据的方法不需要磁盘存储。
- 导入功能更加细粒度：在数据泵技术中，使用 INCLUDE 和 EXCLUDE 参数使得数据泵实用程序可以导入或导出更加细粒度的对象，如可以选择只导出过程或函数等。

4. 数据泵导入导出的目录对象

数据泵作业在数据库服务器上创建所有的备份文件，而 Oracle 要求数据泵必须使用目录对象，以防止用户误操作数据库服务器上特定目录下的操作系统文件。目录对象对应于操作系统上的一个指定目录。

如果当前用户是 DBA 用户，可以使用默认的目录对象而不必再创建数据泵操作的工作目录。此时，数据泵作业会将备份文件、日志文件以及 SQL 文件存储在该目录下。

【实例 25-18】使用数据字典 DBA_DIRECTORIES 查询数据泵作业的目录对象。

```
SQL> conn system/oracle@orcl
已连接。
SQL> col directory_name for a15
SQL>col owner for a10
SQL>col directory_path for a50
SQL> select *
  2  from dba_directories
  3  where directory name ='DATA PUMP DIR';

OWNER      DIRECTORY_NAME  DIRECTORY_PATH
------------------- ------------------------------ ------
SYS        DATA_PUMP_DIR   F:\app\Administraor\admin\orcl\dpdump\
```

使用 SYSTEM 用户登录数据库，查找目录名为 DATA_PUMP_DIR 对应的操作系统文件。该文件对应的目录就是 DBA 用户使用数据泵导出数据时的存储目录。

如果用户欲使用 EXPDP 或者 IMPDP，但是没有可用的目录可用，也不具备创建目录的权限，则提示错误，说明 Oracle 找不到目录对象，无法启动数据泵作业。

【实例 25-19】不具备目录对象的数据泵作业错误。

```
D:\ expdp scott/tiger@orcl

Export: Release11.1.0.6.0 - Production on 星期四, 27 8月, 2009 18:45:08

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
ORA-39002: 操作无效
ORA-39070: 无法打开日志文件。
```

```
ORA-39145：必须指定目录对象参数且不能为空
```

如果用户需要自己创建目录对象，需要具有 CREATE ANY DIRECTORY 权限，如实例 25-20 所示，首先向 SCOTT 用户授权 CREATE ANY DIRECTORY，然后创建属于 SCOTT 用户的数据泵目录对象。

【实例 25-20】向 SCOTT 用户授权并创建目录对象。

```
SQL> conn system/oracle@orcl
已连接。
SQL> grant create any directory to scott;

授权成功。

SQL> conn scott/tiger@orcl
已连接。
SQL> create directory scott_pump_dir as 'f:/scottpumpdir';

目录已创建。
```

下面在数据库服务器上创建数据泵导入导出目录，此时采用 SYSTEM 用户登录，创建目录对象 PUMP_DIR，它对应的操作系统目录为 f:\pump，目的是为下节的使用数据泵实现数据导出做准备。

【实例 25-21】创建数据泵导入导出目录。

```
SQL> create directory pump_dir as 'f:\pump';

目录已创建。
```

此时，创建了一个目录。该目录可以给其他用户使用，但是必须将读、写该目录的权限赋予用户，可以将该目录的读、写权限赋予 SCOTT 用户，如下所示。

```
SQL> grant read on directory pump_dir to scott;

授权成功。
```

一旦授权成功，在使用数据泵导入或导出 SCOTT 用户的数据时，就可以使用该目录作为存储和恢复目录了。

25.4.2 使用数据泵导出数据

数据泵导出 EXPDP 类似于传统的 EXP 实用程序导出数据，使用 EXPDP 允许挂起和恢复作业，并且实现与正在运行的作业交互，可以从正在运行的导出作业中分离，也支持从失败或终止的作业中重启。

1. 数据泵导出参数详解

使用 EXPDP 使得导出时对象选择的粒度更细，并提供并行处理，使用多个数据流导出数据，允许控制导出操作使用的线程数，EXPDP 支持两种数据访问方法，即使用外部表和直接路径访问方法。使用外部表允许数据库服务器从操作系统文件中读取数据，而直接路径方法使用直接路径

API，它很好地改善了导出导入的性能，因为其内部流的数据格式和存储在备份文件中的数据格式相同，减少了数据转换的时间。

EXPDP 提供了三种提取数据的方法，一是只提取数据库中的元数据，即数据库对象的定义，二是只提取数据库中的数据而忽略数据库对象的定义，三是同时提取数据库中的元数据和数据。

在数据泵操作过程中使用 Ctrl+C 组合键时 Oracle 将数据泵操作在后台运行，再将 EXPDP 设置为交互模式，此时后台管理运行的 EXPDP 作业，而用户可以使用同一窗口进行其他操作，如果需要可以切换到 EXPDP 作业。为了方便管理，为每一个作业指定了作业名 JOB_NAME，Oracle 使用这个作业名跟踪作业或重新连接该作业，也可使用默认作业名，在导出作业完成时会给出该名称。

下面分析一下 EXPDP 实用程序的参数。

【实例 25-22】EXPDP 实用程序参数。

```
D:\>expdp help=y

Export: Release 11.1.0.6.0 - Production on 星期四, 27 8月, 2009 19:14:22
Copyright (c) 1982, 2007, Oracle.  All rights reserved.

数据泵导出实用程序提供了一种用于在 Oracle 数据库之间传输
数据对象的机制。该实用程序可以使用以下命令进行调用:

   示例: expdp scott/tiger DIRECTORY=dmpdir DUMPFILE=scott.dmp

您可以控制导出的运行方式。具体方法是: 在 'expdp' 命令后输入
各种参数。要指定各参数, 请使用关键字:

   格式:  expdp KEYWORD=value 或 KEYWORD=(value1,value2,...,valueN)
   示例: expdp scott/tiger DUMPFILE=scott.dmp DIRECTORY=dmpdir SCHEMAS=scott
               或 TABLES=(T1:P1,T1:P2), 如果 T1 是分区表

USERID 必须是命令行中的第一个参数。

关键字                  说明 (默认)
------------------------------------------------------------------------------
ATTACH                   连接到现有作业, 例如 ATTACH [=作业名]。
COMPRESSION               减小有效的转储文件内容的大小
                        关键字值为: (METADATA_ONLY) 和 NONE。
CONTENT                  指定要卸载的数据, 其中有效关键字为:
                        (ALL), DATA_ONLY 和 METADATA_ONLY。
DIRECTORY                供转储文件和日志文件使用的目录对象。
DUMPFILE                 目标转储文件 (expdat.dmp) 的列表,
                        例如 DUMPFILE=scott1.dmp, scott2.dmp, dmpdir:scott3.dmp。
ENCRYPTION_PASSWORD      用于创建加密列数据的口令关键字。
ESTIMATE                 计算作业估计值, 其中有效关键字为:
                        (BLOCKS) 和 STATISTICS。
ESTIMATE_ONLY             在不执行导出的情况下计算作业估计值。
EXCLUDE                  排除特定的对象类型, 例如 EXCLUDE=TABLE:EMP。
FILESIZE                 以字节为单位指定每个转储文件的大小。
FLASHBACK_SCN             用于将会话快照设置回以前状态的 SCN。
```

```
FLASHBACK_TIME        用于获取最接近指定时间的 SCN 的时间。
FULL                  导出整个数据库 (N)。
HELP                  显示帮助消息 (N)。
INCLUDE               包括特定的对象类型, 例如 INCLUDE=TABLE_DATA。
JOB_NAME              要创建的导出作业的名称。
LOGFILE               日志文件名 (export.log)。
NETWORK_LINK          链接到源系统的远程数据库的名称。
NOLOGFILE             不写入日志文件 (N)。
PARALLEL              更改当前作业的活动 worker 的数目。
PARFILE               指定参数文件。
QUERY                 用于导出表的子集的谓词子句。
SAMPLE                要导出的数据的百分比;
SCHEMAS               要导出的方案的列表 (登录方案)。
STATUS                在默认值 (0) 将显示可用时的新状态的情况下,
                      要监视的频率 (以秒计) 作业状态。
TABLES                标识要导出的表的列表 - 只有一个方案。
TABLESPACES           标识要导出的表空间的列表。
TRANSPORT_FULL_CHECK  验证所有表的存储段 (N)。
TRANSPORT_TABLESPACES 要从中卸载元数据的表空间的列表。
VERSION               要导出的对象的版本, 其中有效关键字为:
                      (COMPATIBLE), LATEST 或任何有效的数据库版本。

下列命令在交互模式下有效。
注: 允许使用缩写

命令                  说明
------------------------------------------------------------------------------
ADD_FILE              向转储文件集中添加转储文件。
CONTINUE_CLIENT       返回到记录模式。如果处于空闲状态, 将重新启动作业。
EXIT_CLIENT           退出客户机会话并使作业处于运行状态。
FILESIZE              后续 ADD_FILE 命令的默认文件大小 (字节)。
HELP                  总结交互命令。
KILL_JOB              分离和删除作业。
PARALLEL              更改当前作业的活动 worker 的数目。
                      PARALLEL=<worker 的数目>。
START_JOB             启动/恢复当前作业。
STATUS                在默认值 (0) 将显示可用时的新状态的情况下,
                      要监视的频率 (以秒计) 作业状态。
                      STATUS[=interval]
STOP_JOB              顺序关闭执行的作业并退出客户机。
                      STOP_JOB=IMMEDIATE 将立即关闭
                      数据泵作业。
```

在如上的输出中，对于参数的解释已经很详细，一旦读者尝试过使用 EXPDP 导出数据就很容易掌握和理解这些参数，但是为了使得读者更清晰地理解参数的含义，下面将解释一些经常使用的 EXPDP 指令。

- ATTACH：说明 EXPDP 附加到一个正在运行的现有的 EXPDP 作业。方式为“ATTACH＝JOB_NAME;”。
- CONTENT：说明要导出的数据是元数据还是数据，或者包括元数据和数据，选项包括 ALL、

DATA_ONLY 和 METADATA_ONLY。

- DIRECTORY：说明要导出的备份文件、日志文件和 SQL 文件的存储目录，此时必须事先创建该目录对象，当然可以将其他用户创建的目录对象赋予该当前用户，否则无法启动 EXPDP 程序。
- DUMPFILE：导出的备份文件的文件名，格式为 FILENAME.DMP，如：

```
D:\>exp system/oracle@orcl dumpfile =system_dataonly.dmp
content =data_only
```

- ESTIMATE：计算 EXPDP 导出作业的导出文件的大小，选项包括基于 BLOCKS 或者基于 STATISTICS，其中 BLOCKS 基于数据库块大小的倍数计算备份文件大小，而基于 STATISTICS 使用当前对象的统计量来计算导出的备份文件的大小，如：

```
正在使用 STATISTICS 方法进行估计...
处理对象类型 DATABASE_EXPORT/SCHEMA/TABLE/TABLE_DATA
ORA-39139: 数据泵不支持 XMLSchema 对象。将跳过
TABLE_DATA:"OE"."PURCHASEORDER"。
.  预计为 "SH"."CUSTOMERS"                          9.540 MB
…………
.  预计为 "SYSTEM"."SQLPLUS_PRODUCT_PROFILE"              0 KB
.  预计为 "TSMSYS"."SRS$"                              0 KB
使用 STATISTICS 方法的总估计: 49.08 MB
```

- ESTIMATE_ONLY：在 EXPDP 没有实际地导出作业时估计导出文件的大小，该参数的值为 Y 或 N，如：

```
D:\>expdp system/oracle@orcl directory = pump_dir estimate_only=y

Export: Release 11.1.0.6.0 - Production on 星期四, 27 8 月, 2009 20:33:10

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 -
Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."SYS_EXPORT_SCHEMA_01":  system/********@orcl directory =
pump_dir estimate_only=y
正在使用 BLOCKS 方法进行估计...
处理对象类型 SCHEMA_EXPORT/TABLE/TABLE_DATA
.  预计为 "SYSTEM"."SQLPLUS_PRODUCT_PROFILE"              0 KB
…………
使用 BLOCKS 方法的总估计: 17.37 MB
作业 "SYSTEM"."SYS_EXPORT_SCHEMA_01" 已于 20:33:20 成功完成
```

- EXCLUDE：排除不需要导出的特定对象类型，如 INCLUDE=TABLE:DEPT，对于任何不导出的对象，也不会导出与它有依赖关系的对象，如不导出表，也不会导出和表相关的任

何索引、过程和约束等。

- FLASHBACK_SCN：允许在导出数据库时使用数据库闪回特性，此时 EXPDP 使用规定的 SCN 进行闪回。
- FULL：说明是否导出整个数据库对象，如果该参数为 Y，说明导出数据库的所有对象。
- INCLUDE：说明要导出的特定对象类型，此时会导出该参数指定的对象和与它们有依赖关系的对象。
- JOB_NAME：为了便于管理运行的 EXPDP 作业，设置当前作业的名字。系统默认的命名格式为 sys_operation_mode_nn，如导出 SCOTT 用户的元数据，此时的作业名字为 "SCOTT"."SYS_EXPORT_SCHEMA_01"。
- LOGFILE：说明在导出操作时记录导出过程的日志文件名，其默认名为 export.log，和导出文件保存在相同的目录下，即 directory 参数指定的目录。
- PARALLEL：说明在导出作业时最大的线程数，实现导出作业的并行处理。也可以在作业运行时使用 ATTACH 改变并行度，PARALLEL 参数的默认值为 1，表示使用单线程导出单独个备份文件，如果设置多个工作线程，则要指定相同数量的备份文件，这样多个线程可以同时写多个备份文件。给出一个实例，设置并行度为 2。

```
D:\>expdp system/oracle@orcl directory =pump_dir
 dumpfile =(para_exp01.dmp,para_exp02.dmp) parallel =2
```

- QUERY：允许使用 SQL 语句程序过滤导出的数据，在 Oracle 11g 中，允许使用表名限定 SQL 语句，使得 SQL 语句适用于特定的表，如下所示： QUERY=SCOTT.EMP: “WHERE SAL>3000” 。说明表 EMP 中工资 SAL 大于 3000 的表被导出。
- SCHEMAS：说明要导出数据的模式，该模式列表可以有多个，使用逗号隔开，如果登录的用户不是导出数据的模式，则登录用户必须拥有 exp_full_database 的权限。
- STATUS：该参数在给定的时间间隔内给出作业的状态，该参数以秒为单位，默认值为 0。如下所示：

```
D:\>expdp system/oracle@orcl full=y status = 5 directory = pump_dir
dumpfile=system_full_stauts1.dmp
............
作业：SYS_EXPORT_FULL_01
  操作：EXPORT
  模式：FULL
  状态：EXECUTING
  处理的字节：0
  当前并行度：1
  作业错误计数：0
  转储文件：F:\PUMP\SYSTEM_FULL_STAUTS.DMP
写入的字节：4,096
Worker 1 状态：
  状态：EXECUTING
正在使用 BLOCKS 方法进行估计...
............
Worker 1 状态：
  状态：EXECUTING
```

```
  对象类型:
DATABASE_EXPORT/SCHEMA/TABLE/POST_INSTANCE/PROCACT_INSTANCE
  完成的对象数: 18
  总的对象数: 18
  Worker 并行度: 1
处理对象类型 DATABASE_EXPORT/SCHEMA/TABLE/POST_INSTANCE/PROCDEPOBJ

作业: SYS_EXPORT_FULL_01
  操作: EXPORT
  模式: FULL
  状态: EXECUTING
  处理的字节: 0
  当前并行度: 1
  作业错误计数: 0
  转储文件: F:\PUMP\SYSTEM_FULL_STAUTS1.DMP
    写入的字节: 4,096
```

- TABLES：说明要导出数据库表的列表，此时也会导出与表有依赖关系的对象。
- TABLESPACES：说明要导出的数据库表空间的列表，同时会导出其他表空间中有依赖关系的所有对象。
- VERSION：说明要导出的数据库对象到特定版本的数据库。该参数很好地解决了数据库的对象从高版本迁移到低版本的数据库过程中的版本兼容问题。
- 下面解释交互式命令，数据泵技术的交互方式在正在运行的作业中有效，可以使用交互式命令挂起作业、修改作业参数。交互式命令的说明如下所示。
- ADD_FILE：向导出备份文件集中增加文件以增加目录空间，如在一个作业运行期间输入Ctrl+C组合键切换到交互式导出提示EXPORT。如果该作业因为备份文件的空间不足导致停止，可以使用ADD_FILE命令增加文件到导出目录中：Export>add_file = data_dump_dir:expdata02.dmp。
- STOP_JOB：停止运行的数据泵作业，数据库服务器端的导出数据服务器进程终止。一旦停止一个作业，那么如何重新启动一个数据泵作业呢？此时需要ATTACH命令，如完成SYSTEM用户的全库导出，采用默认的作业名，该名字在执行全库导出时会提示，则停止该作业后，重新打开这个作业的方法如下所示：

```
D:\>expdp system/oracle@orcl attach=system.SYS_EXPORT_FULL_04

Export: Release 11.1.0.6.0 - Production on 星期四, 27 8月, 2009 23:50:59

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release11.1.0.6.0 -
Production
With the Partitioning, OLAP and Data Mining options

作业: SYS_EXPORT_FULL_04
  所有者: SYSTEM
  操作: EXPORT
  创建者权限: FALSE
```

```
GUID: 94DF32382688457CA7085BDFF6A3299E
起始时间: 星期四, 27 8月, 2009 23:51:00
模式: FULL
实例: orcl
最大并行度: 1
EXPORT 个作业参数:
参数名      参数值:
   CLIENT_COMMAND   system/********@orcl full =y directory = pump_dir
dumpfile =export.dmp
状态: IDLING
处理的字节: 0
当前并行度: 1
作业错误计数: 0
转储文件: f:\pump\test_jiaohumoshi_backupwholedb_ofsystem1.dmp
  写入的字节: 229,376
```

- START_JOB：重新恢复由于某种意外导致停止的数据泵作业。
- KILL_JOB：杀死客户机进程和数据泵作业（服务器进程）。
- CONTINUE_CLIENT：退出交互方式（EXPORT 方式）恢复正在运行地导出数据泵作业，实际的数据泵作业不受影响。
- EXIT_CLIENT：停止交互式会话并终止客户机会话，但是实际的数据泵作业不受影响，此时用户可以在当前窗口中继续其他操作。

2. 数据泵导出（EXPDP）数据库实例

下面将通过实例理解如何使用 EXPDP 参数实现数据的导出，使用 EXPDP 可以导出整个数据库、单个模式、特定的表或特定的表空间。以下依次介绍如何实现导出整个数据库或数据库对象。

（1）导出整个数据库

使用 SYSTEM 用户登录数据库，限制备份的数据文件的大小为 100M，一旦备份数据文件满，则自动创建一个新的备份文件，使用了替换变量“%U”来实现备份文件的自动创建，其中 NOLOGFILE=Y，即不记录备份过程。

【实例 25-23】使用 EXPDP 导出整个数据库。

```
F:\>expdp system/oracle@orcl dumpfile = pump_dir:mydb3_%u.dat filesize =
100m nologfile = y
 job_name =tom full = y

Export: Release 11.1.0.6.0 - Production on 星期六, 22 8月, 2009 10:05:25

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."TOM":  system/********@orcl dumpfile = pump_dir:mydb3_%u.dat
filesize = 100m nologfile = y job_name =tom
full = y
正在使用 BLOCKS 方法进行估计...
```

```
处理对象类型 DATABASE_EXPORT/SCHEMA/TABLE/TABLE_DATA
ORA-39139: 数据泵不支持 XMLSchema 对象。将跳过
TABLE_DATA:"OE"."PURCHASEORDER"。
使用 BLOCKS 方法的总估计: 87.5 MB
处理对象类型 DATABASE_EXPORT/TABLESPACE
处理对象类型 DATABASE_EXPORT/PROFILE
............
............
"TSMSYS"."SRS$"                              0 KB       0 行
/卸载了主表 "SYSTEM"."TOM"
*********************************************************************
 的转储文件集为:
MYDB3 01.DAT
"SYSTEM"."TOM" 已经完成, 但是有 1 个错误 (于 10:07:31 完成)
```

上例中备份的文件存储在目录对象 PUMP_DIR 定义的操作系统目录下，此处也可以使用 directory 指令说明目录对象。在 DUMPFILE 参数中指定目录对象使用起来更方便。

（2）导出一个模式

我们导出 SCOTT 模式，在下例中没有 SCHEMA 参数，但是默认导出登录数据库时的模式对象。

【实例 25-24】使用 EXPDP 导出一个 SCOTT 模式。

```
D:\>expdp scott/tiger@orcl dumpfile=pump_dir:scottschema.dmp logfile
=pump_dir:scottschema.log

Export: Release 11.1.0.6.0 - Production on 星期五, 28 8月, 2009 0:09:36

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SCOTT"."SYS_EXPORT_SCHEMA_01":  scott/********@orcl
dumpfile=pump_dir:scottschema.dmp logfile =pump_dir:
a.log
正在使用 BLOCKS 方法进行估计...
处理对象类型 SCHEMA_EXPORT/TABLE/TABLE_DATA
使用 BLOCKS 方法的总估计: 576 KB
处理对象类型 SCHEMA_EXPORT/USER
............
. . 导出了 "SCOTT"."BONUS"                      0 KB       0 行
. . 导出了 "SCOTT"."EMP_TEST2"                   0 KB       0 行
已成功加载/卸载了主表 "SCOTT"."SYS_EXPORT_SCHEMA_01"
**************************************************************************
SCOTT.SYS_EXPORT_SCHEMA_01 的转储文件集为:
  F:\PUMP\SCOTTSCHEMA.DMP
作业 "SCOTT"."SYS_EXPORT_SCHEMA_01" 已于 00:09:52 成功完成
```

（3）导出特定的表

此时使用 TABLES 参数指定导入的表的列表，如果该表不属于登录的用户，但是登录用户有访问这些表的权限，则在 TABLES 参数的表必须使用 schema.tablename 的方式。

【实例 25-25】使用 EXPDP 导入特定的表。

```
F:\>expdp system/oracle@orcl dumpfile = pump_dir:scott_tables_%u.dat
tables=scott.emp,scott.dept nologfile=y job_name=only_scott

Export: Release 11.1.0.6.0 - Production on 星期六, 22 8月, 2009 10:17:33

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."ONLY_SCOTT":  system/********@orcl dumpfile =
pump_dir:scott_tables_%u.dat tables=scott.emp,scott.dept nologfile=
y job_name=only_scott
正在使用 BLOCKS 方法进行估计...
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
使用 BLOCKS 方法的总估计: 128 KB
……………..
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
. . 导出了 "SCOTT"."DEPT"                          5.656 KB       4 行
. . 导出了 "SCOTT"."EMP"                           8.390 KB      28 行
已成功加载/卸载了主表 "SYSTEM"."ONLY_SCOTT"
******************************************************************************
SYSTEM.ONLY_SCOTT 的转储文件集为:
  F:\PUMP\SCOTT_TABLES_01.DAT
作业 "SYSTEM"."ONLY_SCOTT" 已于 10:17:42 成功完成
```

（4）导出表空间

导出指定表空间时可使用 TABESAPCES 参数，如果有多个表空间需要导出，表空间名的使用逗号隔开，这里使用了 PARALLEL 参数，指定数据导出并行线程数量，与之对应使用替换变量“%U”来创建相应数量的备份数据文件，这样每个线程可以独立写一个备份数据文件，提高了导出速度。

【实例 25-26】使用 EXPDP 导出指定的表空间。

```
D:\>expdp system/oracle@orcl dumpfile=pump_dir:users_tbs_
%u.dmp tablespaces=users
filesize=100m parallel=2 logfile=users_tbs.log job_name =exp_users_tbs

Export: Release 11.1.0.6.0 - Production on 星期五, 28 8月, 2009 0:27:18

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."EXP_USERS_TBS":  system/********@orcl
dumpfile=pump_dir:users_tbs_%u.dmp tablespaces=users filesize=100m
parallel=2 logfile=users_tbs.log job_name =exp_users_tbs
正在使用 BLOCKS 方法进行估计...
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
ORA-39139: 数据泵不支持 XMLSchema 对象。将跳过
TABLE_DATA:"OE"."PURCHASEORDER"。
```

```
使用 BLOCKS 方法的总估计: 1.625 MB
. . 导出了 "OE"."LINEITEM_TABLE"                    283.5 KB    2232 行
……….
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
已成功加载/卸载了主表 "SYSTEM"."EXP_USERS_TBS"
******************************************************************************
SYSTEM.EXP_USERS_TBS 的转储文件集为:
  F:\PUMP\USERS_TBS_01.DMP
  F:\PUMP\USERS_TBS_02.DMP
作业 "SYSTEM"."EXP_USERS_TBS" 已经完成, 但是有 1 个错误 (于 00:27:38 完成)
```

（5）只导出数据

使用 EXPDP 的 CONTENT 参数，可以指定导出表数据和元数据（对应参数 ALL），导出表行数据（对应参数 DATA_ONLY）或只导出元数据，即表以及其他数据库对象的定义（对应参数 METADATA_ONLY）。

【实例 25-27】使用 EXPDP 只导出数据行。

```
F:\>expdp system/oracle@orcl dumpfile =pump_dir:mydb_dataonly_
%u.dat filesize=100m
job_name=larry full=y content =data_only logfile
=pump_dir:mydb_exp_dataonly_log
Export: Release 11.1.0.6.0 - Production on 星期六, 22 8月, 2009 10:33:09

Copyright (c) 1982, 2007, Oracle.  All rights reserved.
;;;
连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."LARRY":  system/********@orcl dumpfile
=pump_dir:mydb_dataonly_%u.dat filesize=100m job_name=larry full=y content
=data_only
logfile =pump_dir:mydb_exp_dataonly_log
正在使用 BLOCKS 方法进行估计...
处理对象类型 DATABASE_EXPORT/SCHEMA/TABLE/TABLE_DATA
ORA-39139: 数据泵不支持 XMLSchema 对象。将跳过
TABLE_DATA:"OE"."PURCHASEORDER"。
使用 BLOCKS 方法的总估计: 87.5 MB
. . 导出了 "SYSTEM"."LIN"                          12.75 MB    8373 行
…………
. . 导出了 "TSMSYS"."SRS$"                            0 KB      0 行
已成功加载/卸载了主表 "SYSTEM"."LARRY"
******************************************************************************
SYSTEM.LARRY 的转储文件集为:
  F:\PUMP\MYDB_DATAONLY_01.DAT
作业 "SYSTEM"."LARRY" 已经完成, 但是有 1 个错误 (于 10:34:08 完成)
```

在上例中，设置参数 CONTENTS=DATA_ONLY，说明只导出 SYSTEM 用户中表的数据行，参数 LOGFILE 说明要记录导出过程。

（6）使用参数文件

在使用 EXPDP 导出数据时，由于参数很多,导致每次执行备份都输入一长串指令，这样不但

繁琐而且不易修改，Oracle 的数据泵技术允许使用参数文件，用户事先在参数文件中创建各种参数，保存该文件为 praname.par 文件，然后在执行导出时使用参数 PARFILE 指定参数文件的位置,执行导出备份，这样就不用输入一长串参数指令。

首先建立如下参数文件：

```
directory =pump_dir
dumpfile=para_data_only_%u.dmp
content =data only
exclude =table:"in('salgrade','bonus')"
logfile=para_data_only.log
filesize=50m
parallel=2
job_name =para_data_only
```

保存在文件 EXP.PAR 内，然后在使用 EXPDP 备份数据时，可以使用该参数文件，如下所示：

```
D:\>expdp scott/tiger@orcl parfile=exp.par
```

使用参数 PARFILE 说明参数文件的位置时必须说明绝对路径。

（7）估计空间导出文件的空间大小

EXPDP 使用参数 ESTIMATE_ONLY 计算导出数据所需要的存储空间，显然这是个有用的参数，在导出的数据大小不清楚时，事先知道备份文件的大小，可以提前分配磁盘空间，防止由于磁盘空间不足而引起的 EXPDP 导出作业停止。

【实例 25-28】使用 EXPDP 导出数据时只计算导出作业所需要的空间。

```
F:\>expdp system/oracle@orcl full =y estimate_only=y estimate=statistics
nologfile=y

Export: Release 11.1.0.6.0 - Production on 星期六, 22 8月, 2009 10:38:25

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."SYS_EXPORT_FULL_01":  system/********@orcl full =y
estimate only=y estimate=statistics nologfi
正在使用 STATISTICS 方法进行估计...
处理对象类型 DATABASE_EXPORT/SCHEMA/TABLE/TABLE_DATA
ORA-39139: 数据泵不支持 XMLSchema 对象。将跳过
TABLE_DATA:"OE"."PURCHASEORDER"。
.  预计为 "SH"."CUSTOMERS"                        9.540 MB
............
.  预计为 "SYSTEM"."SQLPLUS_PRODUCT_PROFILE"            0 KB
.  预计为 "TSMSYS"."SRS$"                          0 KB
使用 STATISTICS 方法的总估计: 46.41 MB
作业 "SYSTEM"."SYS_EXPORT_FULL_01" 已经完成 (于 10:38:46 完成)
```

上述计算过程将使用 STATISTIC 的方法计算 SYSTEM 用户所有数据库对象的大小，最后给出一个总的估计结果。

25.4.3 使用数据泵导入数据

Oracle 数据泵导入实用程序（IMPDP）将备份的数据导入到整个数据库、特定的模式、特定的表或者特定的表空间，使用 IMPDP 也可以在不同平台的数据库之间迁移表空间。IMPDP 实用程序通过各种参数支持对数据过滤以及对元数据的过滤，而元数据的过滤可以有效控制要导入的对象类型，如导入表、索引还是授权等。

1. 数据泵导入参数详解

使用数据泵导入实用程序与数据泵导出实用程序一样，可以使用 DIRECTORY、PARFILE、DUMPFILE 和 LOGFILE 等参数，但是正如 ADD_FILE 是数据泵导出 EXPDP 实用程序专有的一样，SQLFILE 参数也是数据泵导入 IMPDP 实用程序所专有的。在执行数据泵导入作业时，往往需要从备份数据文件中途提取 DDL 语句，此时需要使用 SQLFILE。

【实例 25-29】执行数据泵导入作业（使用 SQLFILE 参数）。

```
D:\>impdp system/oracle@orcl directory =pump_dir dumpfile=scott_
tables sqlfile=scott table.sql

Import: Release 11.1.0.6.0 - Production on 星期五, 28 8月, 2009 16:43:20

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已成功加载/卸载了主表 "SYSTEM"."SYS_SQL_FILE_FULL_01"
启动 "SYSTEM"."SYS_SQL_FILE_FULL_01":  system/********@orcl directory
=pump_dir
 dumpfile=scott tables sqlfile=scott table.sql
处理对象类型 TABLE_EXPORT/TABLE/TABLE
处理对象类型 TABLE_EXPORT/TABLE/INDEX/INDEX
处理对象类型 TABLE_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
处理对象类型 TABLE_EXPORT/TABLE/INDEX/STATISTICS/INDEX_STATISTICS
处理对象类型 TABLE_EXPORT/TABLE/TRIGGER
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
作业 "SYSTEM"."SYS_SQL_FILE_FULL_01" 已于 16:43:23 成功完成
```

在执行 IMPDP 导入数据时，IMPDP 实用程序会自动从备份文件（DUMPFILE 参数指定文件）中提取 SQL 语句，并将提取的 SQL 语句保存在参数 SQLFILE 指定的文件中。整个导入过程中创建的 scott_table.sql 文件保存在参数 DIRECTORY 指定的目录对象中，SQLFILE 参数的作用是为特定的备份文件提取 SQL 的 DDL 语句，而不是产生实际的 DDL 行为，它提取出备份文件中的 DDL 的 SQL 脚本内容，使得用户知道在备份文件中发生了哪些 DDL 行为。

【实例 25-30】SCOTT_TABLE.SQL 文件中的部分记录结果。

```
-- CONNECT SYSTEM
-- new object type path is: TABLE EXPORT/TABLE/TABLE
CREATE TABLE "SCOTT"."DEPT"
```

```
   ( "DEPTNO" NUMBER(2,0),
    "DNAME" VARCHAR2(14),
    "LOC" VARCHAR2(13)
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

CREATE TABLE "SCOTT"."EMP"
   ( "EMPNO" NUMBER(4,0),
    "ENAME" VARCHAR2(10),
    "JOB" VARCHAR2(9),
    "MGR" NUMBER(4,0),
    "HIREDATE" DATE,
    "SAL" NUMBER(7,2),
    "COMM" NUMBER(7,2),
    "DEPTNO" NUMBER(2,0)
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

-- new object type path is: TABLE_EXPORT/TABLE/INDEX/INDEX
-- CONNECT SCOTT
CREATE UNIQUE INDEX "SCOTT"."PK_DEPT" ON "SCOTT"."DEPT" ("DEPTNO")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" PARALLEL 1 ;

  ALTER INDEX "SCOTT"."PK_DEPT" NOPARALLEL;
............
```

上述的 SQL 脚本显示了导入备份文件过程中要创建表、索引和各种触发器。

数据泵导入 IMPDP 和数据泵导出 EXPDP 一样，需要通过各种参数来控制导入行为，如使用 CONTENT 参数确定是否导入元数据（METADATA_ONLY）、行数据（DATA_ONLY），还是行数据和元数据（ALL），DUMPFILE 参数确定导入文件名，LOGFILE 参数确定导入过程的日志文件等。

使用 IMPDP help=y 指令可以查看 IMPDP 指令的所有可选参数和基本含义。

【实例 25-31】查看 IMPDP 导入程序的所有参数。

```
D:\>IMPDP HELP=Y

Import: Release 11.1.0.6.0 - Production on 星期五, 28 8月, 2009 17:13:22

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


数据泵导入实用程序提供了一种用于在 Oracle 数据库之间传输
```

```
数据对象的机制。该实用程序可以使用以下命令进行调用:

    示例: impdp scott/tiger DIRECTORY=dmpdir DUMPFILE=scott.dmp

您可以控制导入的运行方式。具体方法是: 在 'impdp' 命令后输入
各种参数。要指定各参数, 请使用关键字:

    格式:  impdp KEYWORD=value 或 KEYWORD=(value1,value2,...,valueN)
    示例: impdp scott/tiger DIRECTORY=dmpdir DUMPFILE=scott.dmp

USERID 必须是命令行中的第一个参数。

关键字                  说明 (默认)
------------------------------------------------------------------------------
ATTACH                   连接到现有作业, 例如 ATTACH [=作业名]。
CONTENT                  指定要加载的数据, 其中有效关键字为:
                         (ALL), DATA_ONLY 和 METADATA_ONLY。
DIRECTORY                供转储文件, 日志文件和 sql 文件使用的目录对象。
DUMPFILE                 要从 (expdat.dmp) 中导入的转储文件的列表,
                         例如 DUMPFILE=scott1.dmp, scott2.dmp, dmpdir:scott3.dmp。
ENCRYPTION_PASSWORD      用于访问加密列数据的口令关键字。
                         此参数对网络导入作业无效。
ESTIMATE                 计算作业估计值, 其中有效关键字为:
                         (BLOCKS) 和 STATISTICS。
EXCLUDE                  排除特定的对象类型, 例如 EXCLUDE=TABLE:EMP。
FLASHBACK_SCN            用于将会话快照设置回以前状态的 SCN。
FLASHBACK_TIME           用于获取最接近指定时间的 SCN 的时间。
FULL                     从源导入全部对象 (Y)。
HELP                     显示帮助消息 (N)。
INCLUDE                  包括特定的对象类型, 例如 INCLUDE=TABLE_DATA。
JOB_NAME                 要创建的导入作业的名称。
LOGFILE                  日志文件名 (import.log)。
NETWORK_LINK             链接到源系统的远程数据库的名称。
NOLOGFILE                不写入日志文件。
PARALLEL                 更改当前作业的活动 worker 的数目。
PARFILE                  指定参数文件。
QUERY                    用于导入表的子集的谓词子句。
REMAP_DATAFILE           在所有 DDL 语句中重新定义数据文件引用。
REMAP_SCHEMA             将一个方案中的对象加载到另一个方案。
REMAP_TABLESPACE         将表空间对象重新映射到另一个表空间。
REUSE_DATAFILES          如果表空间已存在, 则将其初始化 (N)。
SCHEMAS                  要导入的方案的列表。
SKIP_UNUSABLE_INDEXES 跳过设置为无用索引状态的索引。
SQLFILE                  将所有的 SQL DDL 写入指定的文件。
STATUS                   在默认值 (0) 将显示可用时的新状态的情况下,
                         要监视的频率 (以秒计) 作业状态。
STREAMS_CONFIGURATION 启用流元数据的加载
TABLE_EXISTS_ACTION      导入对象已存在时执行的操作。
                         有效关键字: (SKIP), APPEND, REPLACE 和 TRUNCATE。
TABLES                   标识要导入的表的列表。
```

```
TABLESPACES              标识要导入的表空间的列表。
TRANSFORM                要应用于适用对象的元数据转换。
                         有效的转换关键字: SEGMENT_ATTRIBUTES, STORAGE
                         OID 和 PCTSPACE。
TRANSPORT_DATAFILES      按可传输模式导入的数据文件的列表。
TRANSPORT_FULL_CHECK     验证所有表的存储段 (N)。
TRANSPORT_TABLESPACES    要从中加载元数据的表空间的列表。
                         仅在 NETWORK_LINK 模式导入操作中有效。
VERSION                  要导出的对象的版本, 其中有效关键字为:
                         (COMPATIBLE), LATEST 或任何有效的数据库版本。
                         仅对 NETWORK_LINK 和 SQLFILE 有效。

下列命令在交互模式下有效。
注: 允许使用缩写

命令                     说明 (默认)
------------------------------------------------------------------------------
CONTINUE_CLIENT          返回到记录模式。如果处于空闲状态, 将重新启动作业。
EXIT_CLIENT              退出客户机会话并使作业处于运行状态。
HELP                     总结交互命令。
KILL_JOB                 分离和删除作业。
PARALLEL                 更改当前作业的活动 worker 的数目。
                         PARALLEL=<worker 的数目>。
START_JOB                启动/恢复当前作业。
                         START_JOB=SKIP_CURRENT 在开始作业之前将跳过
                         作业停止时执行的任意操作。
STATUS                   在默认值 (0) 将显示可用时的新状态的情况下,
                         要监视的频率 (以秒计) 作业状态。
                         STATUS[=interval]
STOP_JOB                 顺序关闭执行的作业并退出客户机。
                         STOP_JOB=IMMEDIATE 将立即关闭
                         数据泵作业。
```

下面对 IMPDP 导入程序的参数进行适当分类，使得读者易于理解和掌握，然后再详细介绍这些参数的含义和使用注意事项。

（1）目录和文件相关的参数

对目录和文件相关的参数说明如下。

- DIRECTORY：说明备份文件、日志文件和 SQL 文件的目录对象，如果没有定义目录，则会使用 PUMP_DIR 的默认值。
- DUMPFILE：说明备份文件名，如导入数据时需要多个备份文件，则用逗号分隔这些文件名，在 DUMPFILE 参数后可以使用目录（如 DUMFILE=PUMP_DIR:BACKUP.DMP），也可以使用替换变量（%U）告诉 IMPDP 可以使用多个备份文件，如 DUMPFILE=PUMP_DIR:BACKUP_%U.DMP。
- PARFILE：说明参数文件，IMPDP 使用外部定义了一个参数文件执行导入行为，该参数文件是本地的，使用时需要告诉 IMPDP 参数文件的绝对位置，如 D:\IMPDP SYSTEM/ORACLE@ORCL PARFILE=D:\PAR\EXP.PAR。

- LOGFILE：说明使用日志文件保存导入过程的信息，该参数的值是日志文件的名字，如 LOGFILE =MYLOG.LOG。
- NOLOGFILE：说明不使用日志文件记录导入过程，如 NOLOGFILE=Y。
- SQLFILE：说明从备份文件中提取 SQL 的 DDL 语句，并写入该参数设置的文件中，如 SQLFILE=MYSQLFILE.SQL。该文件默认保存在 DIRECTORY 参数设置的目录对象中。

（2）过滤参数

对过滤参数的说明如下。

①INCLUDE

说明要导入的特定对象，如只导入表，此时会导入和导入特定对象有依赖关系的对象，如索引、触发器等。

下面是使用 INCLUDE 参数的实例，说明只允许导入表对象，且只有两个表可以导入。

```
INCLUDE=TABLE: "IN ('EMP','DEPT') "
```

也可以使用 QUERY 参数过滤要导入的表数据，此时数据泵导入作业使用外部表数据方法访问数据，而不是采用直接路径方法，如下所示。

```
INCLUDE=TABLE: "IN ('EMP','DEPT') "
QUERY=EMP: "WHERE sal>3000 ORDER BY sal"
```

②TABLE_EXITS_ACTION

该参数说明当导入的表已经存在时，IMPDP 导入程序的行为，参数 TABLE_EXITS_ACTION 有 4 个值，SKIP 表示如果该表存在则跳过该表，它是默认值；APPEND 将导入的数据行附加到当前存在的表中；TRUNCATE 截断表并从导入数据文件中重新装载数据；REPLACE 删除存在的表然后重建该表并导入数据。

③EXCLUDE

在导入操作中排除特定的元数据，如不导入特定的表，此时也不会导入和排除对象有依赖关系的其他对象。如下所示告诉 IMPDP 程序不导入表 EMP 和 DEPT：

```
EXCLUDE=TABLE: "IN ('EMP','DEPT') "
```

（3）导入作业参数

①JOB_NAME

说明导入作业名，IMPDP 提供了很多可管理性，如停止作业和恢复作业，附加（ATTACH）到特定的作业，都需要作业名来关联导入作业。

②PRALLEL

说明当前导入作业的线程数，该值的默认值为 1。

③STATUS

监视导入作业的状态频率，该参数的默认值为 0。

（4）导入方式参数

①TABLES

说明允许导入指定的表，如果有多个表使用逗号分隔，同时也导入与这些表有依赖关系的对

象，如索引、触发器和函数等。

②SCHEMAS

说明要导入的模式列表，要使用该参数登录数据库的用户必须拥有 imp_full_database 的权限。

③TABLESPACES

说明要导入的表空间的列表，在导入这些表空间的同时也要求导入与表空间有依赖关系的所有数据库对象。

④FULL

说明要导入整个数据库。该参数的默认值为 n。

（5）重新映射参数

重新映射使得在数据导入过程中将数据从一个数据库对象移动到另一个数据库对象，可以映射模式、映射数据文件和映射表空间，映射可以理解为“数据对象移动”。

①REMAP_SCHEMA

重新映射模式，可以将对象从一个模式移动到另一个模式，代码如下：

```
D:\>impdp system/oracle@orcl dumpfile =pump dir:SCHEMA SCOTT.DMP
 remap_schema=scott:linzi
```

上例将 SCOTT 模式下的所有数据库对象移动到 LINZI 模式下，这样使用 LINZI 模式登录数据库，就可以使用 SCOTT 用户的所有数据库对象。通过实例 25-32 验证重映射模式的结果。

【实例 25-32】验证模式 LINZI 中是否有 SCOTT 模式对象。

```
SQL> conn linzi/linzi@orcl
已连接。
SQL> desc dept;
 名称                                        是否为空？ 类型
 ----------------------------------------------------------- ---
 DEPTNO                                       NOT NULL NUMBER(2)
 DNAME                                                 VARCHAR2(14)
 LOC                                                   VARCHAR2(13)
```

可见 SCOTT 用户的 DEPT 表对象在 LINZI 模式下可以直接使用，说明模式迁移成功。

②REMAP_DATAFILE

在导入数据时，重新定义数据文件的名称和目录，如下所示：

```
D:\impdp  system/oracle@orcl  directory=pump_dir  dumpfile=backup_full.dmp
remap_datafile='c:\mydb.dbf':'d:\mydb\newdb.dbf
```

③REMAP_TABLESPACE

重映射表空间使得将数据对象从一个表空间移动到另一个表空间，如下所示：

```
D:\impdp system/oracle@orcl remap_tablespace='users':'mynewusers'
 directory=pump_dir dumpfile=backup_full.dmp
```

（6）转换参数

该参数说明在导入数据泵作业时可以选择导入某个对象的存储参数或其他属性值，如导入表时，不导入该表的存储属性等。

TRANSFORM 参数的语法如下所示：

```
TRANSFORM= transform_name:value[:object_type]
```

下面介绍各参数的含义。

- transform_name：转换名由 4 个选项组成，代表 4 种基本的对象特征。其中 SEGMENT_ATTRIBUTES 段属性包括物理属性、存储参数、表空间等，该参数的值为 Y 或 N，如果选择 SETMENT_ATTRIBUTES=Y 则说明导入作业包括对象的这些属性；STORAGE 存储属性说明是否导入对象的存储属性，如果 STORAGE=Y 说明对象的存储属性作为导入作业的一部分。OID 说明是否分配新的 OID 给对象表。PCTSPACE：提供一个正数值，可以增加对象的分配尺寸。
- value：转换名的值中前三个值，即 SEGMENT_ATTRIBUTES、STORAGE 和 OID，默认值为 Y，说明默认数据泵导入对象的存储属性和段属性。而 PCTSPACE 取一个正数值。
- object_type：对象类型说明需要转换哪些类型的对象，这些类型包括表、索引、表空间以及约束等。

【实例 25-33】说明如何使用 TRANSFORM 参数。

```
D:\>impdp system/oracle@orcl tables =emp directory =pump_
dir dumpfile=scott tables.dmp transform=segment attributes:n:table

Import: Release 11.1.0.6.0 - Production on 星期五, 28 8月, 2009 21:07:29

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已成功加载/卸载了主表 "SYSTEM"."SYS_IMPORT_TABLE_01"
启动 "SYSTEM"."SYS_IMPORT_TABLE_01":  system/********@orcl tables =
emp directory =pump_dir dumpfile=scott_tables.dmp tra
nsform=segment_attributes:n:table
处理对象类型 TABLE_EXPORT/TABLE/TABLE
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
. . 导入了 "SCOTT"."EMP"                          9.523 KB      56 行
处理对象类型 TABLE_EXPORT/TABLE/TRIGGER
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
作业 "SYSTEM"."SYS_IMPORT_TABLE_01" 已于 21:07:31 成功完成
```

在上例中，事先删除 SCOTT 用户的表 EMP，模拟一个故障，此时再使用备份文件 scott_tables.dmp（该文件备份了 SCOTT 用户的 DEPT 和 EMP 表）恢复表 EMP，但是不需要该表的 SEGMENT_ATTRIBUTES 属性。

（7）闪回参数

①FLASHBACK_SCN

使用 Oracle 的闪回特性，允许导入和闪回 SCN 接近的数据。

②FLASHBACK_TIME

使用 Oracle 的闪回特性，允许导入和指定闪回时间接近的数据。

（8）与可移植表空间有关的参数

Oracle 的可移植表空间使得将数据从一个数据库移动到另一个数据库非常容易，可以方便地将一个数据库表空间中的数据迁移到其他数据库中的表空间中。

①TRANSPORT_TABLESPACES

说明要迁移的表空间列表，如下所示：

```
D:\>expdp system/oracle@orcl directory=pump
dir dumpfile=users_tbs_metadata.dmp transport_tablespaces=users
```

要迁移的表空间必须置于只读状态，但是导出备份文件 users_tbs_metadata.dmp 只包含表空间 USERS 的元数据。

②TRANSPORT_FULL_CHECK

迁移表空间时，检查迁移表空间内的对象与迁移表空间外的对象是否具有依赖性，该参数只有在使用 NETWORK_LINK 参数时才有效。

③TRANSPORT_DATAFILES

在执行表空间导入时，目标数据库将使用源数据库中拷贝过来的数据文件作为可以移植表空间的数据文件。该参数说明数据文件名。

（9）交互模式参数

数据泵导入的交互参数和数据泵导出的交互参数的功能是一样的，在数据泵导入参数中没有 ADD_FILE 参数，它只对数据泵导出程序有效，而其他参数以及切换到交互模式数据泵的导入与导出是一样的。

- PARALLEL：说明当前作业的活跃 WORKER 数量。
- CONTINUE_CLIENT：在切换到交互模式后，返回记录模式。
- EXIT_CLIENT：退出客户登录模式，但是不终止导入作业。
- KILL_JOB：分离或删除当前导入作业。
- START_JOB：在导入作业被意外终止后，可以重启或恢复当前作业。
- STATUS：监视当前导入作业的状态，该参数是一个整数值，默认值为 0，如果设置 STATUS=5，说明每 5 秒钟刷新一次导入作业的状态信息。
- STOP_JOB：关闭当前执行的作业并退出客户端，如果有多个导入作业，则顺序关闭这些作业。如果设置 STOP_JOB=IMMEDIATE 将立即关闭数据泵作业。

2. 数据泵导入数据库实例

使用数据泵导入 IMPDP 可以导入基于使用数据泵导出的备份文件，可以导入整个数据库、指定的表空间、指定的表或者指定的数据库对象类型，如索引、函数、存储过程和触发器等。下面通过实例依次说明如何使用数据泵导入作业。

（1）导入整个数据库

导入整个数据库至少需要两个参数，一个是 FULL，设置 FULL=Y 说明是导入全库，一个是 DUMPFILE，说明要导入的备份文件的目录和名称，当然最好设置 JOB_NAME 参数，因为它允许切换到交换模式，允许终止或重启导入会话，如下所示。

【实例 25-34】使用 IMPDP 导入整个数据库。

```
D:\>impdp system/oracle@orcl dumpfile=pump_dir:full_db_%u.dat logfile=
myfulldb.log parallel= 3 job_name=my_fulldb_impdp full =y
```

在上例中，导入的备份文件位于目录对象 PUMP_DIR 定义的操作系统目录下，这里使用了替换变量“%U”说明，它的值为 01~99，IMPDP 程序将依次读取备份文件集中的多个备份文件。参数 LOGFILE=MYFULLDB.LOG 记录数据导入过程，PARALLEL=3 说明启动三个线程完成数据导入，JOB_NAME 为 my_fulldb_impdp，关键是 FULL=Y 说明是导入整个数据库，如果不使用 FULL 参数，默认导入模式 SYSTEM 的所有数据库对象。

（2）导入表空间

使用 IMPDP 导入特定的表空间时，需要有备份表空间文件，需要使用 TABLESPACES 参数说明要导入的表空间名，此时实际上是导入表空间中的所有数据库对象，当然这些工作都由 IMPDP 自己操作完成，由于表空间中有表对象，如果当前的数据库中的表空间已有相应的表对象，则最好告诉 IMPDP 该怎么做，此时需要参数 TABLE_EXITS_ACTION，它的默认值为 SKIP，即如果表已经存在则跳过。建议使用 REPLACE 或 TRUNCATE，前者表示重建表，后者表示删除掉当前表中的数据，然后使用备份文件中的表数据进行加载，但是会跳过所有相关元数据。

【实例 25-35】使用 IMPDP 导入特定的表空间。

```
D:\>impdp system/oracle@orcl
dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT logfile= tablespaces=users
table_exists_action=replace

Import: Release 11.1.0.6.0 - Production on 星期六, 29 8月, 2009 10:59:40

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已成功加载/卸载了主表 "SYSTEM"."SYS_IMPORT_TABLESPACE_01"
启动 "SYSTEM"."SYS_IMPORT_TABLESPACE_01":  system/********@orcl
dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT nologfi
le=y tablespaces=users table_exists_action=replace
处理对象类型 TABLE_EXPORT/TABLE/TABLE
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
. . 导入了 "LINZI"."CREATE$JAVA$LOB$TABLE"          5.835 KB       1 行
. . 导入了 "OE"."CREATE$JAVA$LOB$TABLE"             5.828 KB       1 行
…………
. . 导入了 "SCOTT"."EMP_TEST2"                          0 KB       0 行
处理对象类型 TABLE_EXPORT/TABLE/INDEX/INDEX
处理对象类型 TABLE_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
```

```
处理对象类型 TABLE_EXPORT/TABLE/INDEX/STATISTICS/INDEX_STATISTICS
处理对象类型 TABLE_EXPORT/TABLE/CONSTRAINT/REF_CONSTRAINT
处理对象类型 TABLE_EXPORT/TABLE/TRIGGER
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
作业 "SYSTEM"."SYS_IMPORT_TABLESPACE_01" 已于 11:00:01 成功完成
```

上述代码使用了 LOGFILE 参数，这是一个好习惯，因为在备份后可以使用日志文件查询备份过程和备份的所有数据库对象信息。TABLESPACES 参数是必须的，它说明要导入备份文件中的那个表空间。参数 TABLE_EXISTS_ACTION=REPALCE 说明遇到已经存在的表要重建该表对象，然后使用备份文件中的数据进行加载。

（3）导入指定的表

使用 IMPDP 导入特定的表时可利用 TABLES 参数实现，该参数后是要导入的表对象的列表，如果有多个表时，可使用逗号分隔。设置 TABLE_EXISTS_ACTION=REPLACE，如果该表存在则先删除，而后再加载数据。

【实例 25-36】导入特定的表对象。

```
D:\>impdp scott/tiger@orcl dumpfile=pump dir:MYDB TBS USERSANDSYSTEM
01.DAT nologfile=y tables= emp table_exists_action=replace

Import: Release 11.1.0.6.0 - Production on 星期六, 29 8月, 2009 11:05:04

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
ORA-39154: 外部方案中的对象已从导入中删除
已成功加载/卸载了主表 "SCOTT"."SYS_IMPORT_TABLE_01"
启动 "SCOTT"."SYS_IMPORT_TABLE_01":  scott/********@orcl
dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT nologfile=y ta
bles= emp table_exists_action=replace
处理对象类型 TABLE_EXPORT/TABLE/TABLE
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
. . 导入了 "SCOTT"."EMP"                          9.523 KB      56 行
处理对象类型 TABLE_EXPORT/TABLE/TRIGGER
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
作业 "SCOTT"."SYS_IMPORT_TABLE_01" 已于 11:05:08 成功完成
```

上例中，导入了表 EMP，注意此时使用 SCOTT 用户登录，所以只导入 SCOTT 用户中的 EMP 表，如果使用 SYSTEM 用户登录就会将 EMP 表导入多个用户，因为 SYSTEM 用户拥有全部的数据库权限。

【实例 25-37】使用 SYSTEM 用户登录数据库导入 EMP 表。

```
D:\>impdp system/oracle@orcl
dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT
 nologfile=y tables= emp table_exists_action=replace
…………
处理对象类型 TABLE_EXPORT/TABLE/TABLE
```

```
处理对象类型 TABLE_EXPORT/TABLE/TABLE_DATA
. . 导入了 "LINZI"."EMP"                          9.523 KB      56 行
. . 导入了 "OE"."EMP"                            9.515 KB      56 行
. . 导入了 "SCOTT"."EMP"                          9.523 KB      56 行
处理对象类型 TABLE_EXPORT/TABLE/TRIGGER
处理对象类型 TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
```

显然此时导入数据时，IMPDP 将在数据库中搜索用户 EMP 表，发现当前数据库中用户 LINZI、OE 和 SCOTT 都拥有该表，所以将它可以“管理”的所有 EMP 表进行了恢复。

（4）导入指定的数据库对象

导入指定的数据库对象可使用 INCLUDE 参数实现，从备份文件中恢复 SCOTT 用户的所有表和触发器对象，而对于已经存在的表，则重建后再加载数据，例如：

```
D:\>impdp scott/tiger@orcl dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT
nologfile=y include=table,trigger table_exists_action=replace
```

在备份文件 MYDB_TBS_USERSANDSYSTEM_01.DAT 中，备份了两个表空间，即 USERS 和 SYSTEM 中的所有数据库对象，而 SCOTT 用户的数据库对象就存放在这两个表空间中，读者在试验时，可以自己先使用 EXPDP 程序做备份文件，然后再使用 IMPDP 导入特定数据库对象，注意使用哪个用户登录就导入哪个用户的数据库对象。使用 LINZI 用户登录（笔者自建的用户）的示意代码如下：

```
D:\>impdp linzi/linzi@orcl dumpfile=pump_dir:MYDB_TBS_USERSANDSYSTEM_01.DAT
nologfile=y include=table,trigger table_exists_action=replace
………..
. . 导入了 "LINZI"."CREATE$JAVA$LOB$TABLE"          5.835 KB       1 行
. . 导入了 "LINZI"."DEPT"                         5.656 KB       4 行
. . 导入了 "LINZI"."DEPT_TEST"                    5.789 KB      10 行
. . 导入了 "LINZI"."EMP"                          9.523 KB      56 行
. . 导入了 "LINZI"."EMP_TEST3"                    6.367 KB       4 行
. . 导入了 "LINZI"."ROOMS"                        6.062 KB       8 行
. . 导入了 "LINZI"."SALGRADE"                     5.585 KB       5 行
. . 导入了 "LINZI"."USER_MODIFY_TABLE"             7.390 KB      68 行
. . 导入了 "LINZI"."BACKUP_DELETE_EMP_TABLE"           0 KB       0 行
. . 导入了 "LINZI"."BONUS"                             0 KB       0 行
. . 导入了 "LINZI"."EMP_TEST2"                         0 KB       0 行
……
```

上例只给出了部分输出结果，其他和上例相同，通过这个实例是要告诉读者使用 INCLUDE 过滤了要导入的对象，而登录用户决定了要恢复哪个用户的数据库对象。

25.4.4 使用数据泵迁移表空间

Oracle 提供了可迁移表空间的新特性，它使得数据库之间移动数据既快速又简单，尤其对于移动大对象数据，使用迁移表空间特性需要属于源数据库的两个文件，一个是要迁移的表空间的所有数据文件，另一个是使用 EXPDP 程序导出的表空间的元数据。将这两类文件拷贝到目标数据库上，再使用 IMPDP 程序执行导入迁移表空间。

在迁移表空间时需要几个必备的步骤：

01 确定要迁移的表空间，并验证它是否与其他表空间中的对象有依赖关系。
02 导出迁移表空间的元数据，使用 EXPDP 程序生成一个.DMP 表空间元数据备份文件。
03 把备份文件的元数据文件和迁移表空间中的数据文件拷贝到目录数据库。
04 在目标数据库上执行迁移表空间。

下面详细介绍这 4 个步骤的实现。

1. 选择要迁移的表空间

要迁移的表空间必须满足是自包含的，即该表空间中的对象不能与其他表空间中的对象有依赖关系，所以需要实现验证，Oracle 提供了一个方法，该方法在 DBMS_TTS 程序包中，该方法是 TRANSPORT_SET_CHECK(tbs_name,boolean)。验证方法如实例 25-38 所示。

【实例 25-38】验证表空间是否是自包含。

```
SQL> execute sys.dbms tts.transport set check('users',true);

PL/SQL 过程已成功完成。
```

在上例中，使用 EXECUTE 执行过程来验证要迁移的表空间 USERS 是否是自包含的。过程 TRANSPORT_SET_CHECK 没有返回错误消息，说明迁移表空间 USERS 是自包含的，可以作为迁移表空间使用。

2. 导出要迁移表空间的元数据

在将表空间迁移到目标数据库之前，必须使用 EXPDP 导出程序创建迁移表空间的元数据集。而操作之前必须将要迁移的表空间置为只读状态。

【实例 25-39】将表空间置于只读状态。

```
SQL> alter tablespace users read only ;

表空间已更改。
```

要迁移的表空间被置于只读状态后，就可以使用数据泵导出程序 EXPDP 为表空间 USERS 创建元数据备份文件，如实例 25-40 所示。

【实例 25-40】导出迁移表空间的目录元数据。

```
D:\>expdp system/oracle@orcl dumpfile=pump_dir:transport_users_tbs.dmp
transport_tablespaces=users
Export: Release 11.1.0.6.0 - Production on 星期六, 29 8 月, 2009 11:50:31
Copyright (c) 1982, 2007, Oracle.  All rights reserved.
连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
启动 "SYSTEM"."SYS_EXPORT_TRANSPORTABLE_01":  system/********@orcl
dumpfile=pump_dir:transport_users_tbs.dmp transport_tablespaces=users
处理对象类型 TRANSPORTABLE_EXPORT/PLUGTS_BLK
............
```

```
处理对象类型 TRANSPORTABLE_EXPORT/POST_INSTANCE/PLUGTS_BLK
已成功加载/卸载了主表 "SYSTEM"."SYS_EXPORT_TRANSPORTABLE_01"
*************************************************************************
SYSTEM.SYS_EXPORT_TRANSPORTABLE_01 的转储文件集为:
  F:\PUMP\TRANSPORT_USERS_TBS.DMP
作业 "SYSTEM"."SYS_EXPORT_TRANSPORTABLE_01" 已经完成, 但是有 1 个错误 (于
 11:50:47 完成)
```

把表空间 USERS 的元数据备份到目录对象 PUMP_DIR 指定的操作系统目录下，因为导出元数据只是导出表空间中数据库对象的定义，而不是数据行，所以这个导出过程很快。本例中只导出表空间 USERS 的元数据定义，所以文件很小，只有大约 700K。

3. 将备份文件和迁移表空间中的数据文件拷贝到目录数据库

在创建了表空间元数据备份文件后，需要拷贝要迁移的表空间中的所有数据文件和刚才创建的表空间元数据备份文件到目标数据库的一个可访问目录下。此时可以使用任意的拷贝方式，使用网络传输或者刻成光盘等。

4. 在目标库上导入可迁移的表空间

在笔者的计算机上，将表空间的元数据备份文件和表空间中的所有数据文件都拷贝到目标数据库所在计算机的一个磁盘上，保存目录为 F:\PUMP，下面就可以运行 IMPDP 程序在目标数据库中导入源表空间的元数据，目标数据库将使用源表空间中拷贝过来的所有数据文件作为迁移表空间的数据文件。

【实例 25-41】在目标数据库中导入迁移表空间。

```
D:\>impdp system/oracle@orcl dumpfile=transport users tbs.dmp
transport_datafiles='users01.dbf' directory=pump_dir
Import: Release 11.1.0.6.0 - Production on 星期六, 29 8月, 2009 15:41:27
Copyright (c) 1982, 2007, Oracle.  All rights reserved.
连接到: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP and Data Mining options
已成功加载/卸载了主表 "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01"
启动 "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01":  system/********@orcl
dumpfile=transport_users_tbs.dmp transport_datafiles=
'users01.dbf' directory=pump_dir
处理对象类型 TRANSPORTABLE_EXPORT/CONSTRAINT/CONSTRAINT
………..
作业 "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01" 已经完成(于 15:41:29 完成)
```

在目标数据库中迁移表空间时，使用 IMPDP 从备份数据中导出迁移表空间的元数据，并创建各种数据库对象，如触发器、表和约束等，但是没有导入任何数据，因为数据已经在拷贝的数据文件中了，使用 TRANSPORT_DATAFILES 参数说明了需要的参数，而迁移的表空间已经在目标数据库上了，至此已成功迁移了表空间。

25.5　备份与恢复

数据库备份是 DBA 日常工作的重要部分，本节将讲解冷备份的两种方式，以及相应的数据恢复方式。

25.5.1　脱机备份方法

用户管理的脱机备份是指先关闭数据库，而后使用操作系统工具拷贝数据库文件，即数据文件、控制文件和重做日志文件，这种备份总是一致的数据库备份，因为此时用户无法访问数据库，没有数据的变化，备份后的数据库和当前的数据库中的数据是一致的。用户管理的脱机备份遵循以下步骤。

01 首先确认数据库文件所在的操作系统目录。记录下这些文件的目录信息。
02 关闭数据库，此时不要使用 SHUTDOWN ABORT 关闭数据库。
03 拷贝数据库文件到指定的备份目录中。
04 重启启动数据库，完成备份。

下面演示用户管理的脱机备份的具体过程，读者只要按照步骤操作，能很容易完成一次脱机备份。

1. 查看数据库文件所在的目录并记录目录信息

这些数据库文件包括数据文件，控制文件和重做日志文件。

【实例 25-42】查看数据文件的存储目录。

```
SQL> col file_name for a55
SQL> select file name, tablespace name
  2* from dba_data_files

FILE_NAME                                              TABLESPACE_NAME
----------------------------------------------------------------------
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF      USERS
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF     SYSAUX
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF    UNDOTBS1
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF     SYSTEM
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF    EXAMPLE
```

所有的数据文件都位于一个目录下，在 Oracle 11g 中多了辅助表空间 SYSAUX 表空间中的数据，如果读者安装了 Oracle 9i 数据库，则不会看到该表空间。

【实例 25-43】查看控制文件的存储目录。

```
SQL> select name from v$controlfile;

NAME
-------------------------------------------------------------------------
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL01.CTL
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL02.CTL
```

```
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL03.CTL
```

在笔者的计算机上控制文件位于同一个目录下，而且该目录是 Oracle 默认的控制文件目录，但实际中这样做是很不安全的，需要实现控制文件的不同磁盘的冗余分布。

【实例 25-44】查看重做日志文件的存储目录。

```
SQL> select member
  2  from v$logfile;

MEMBER
--------------------------------------------------------------------------------
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO03.LOG
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO02.LOG
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO01.LOG
```

显然重做日志也存放在同一个目录下，和控制文件一样这样的做法也是不安全的，在生产数据库中需要设置多个重做日志组，并且每个日志组有多个重做日志成员，且这些日志成员分布在不同的磁盘上，以提高系统的可靠性。

2. 关闭 Oracle 数据库

【实例 25-45】关闭 Oracle 数据库。

```
SQL> conn system/oracle@orcl as sysdba
已连接。
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
```

如果使用 SHUTDOWN ABORT 有可能造成数据丢失，所以为了快速关闭数据库，最好使用 SHUTDOWN IMMEDIATE 安全地关闭数据库。

3. 拷贝数据文件到备份目录

因为笔者的计算机上所有的数据库文件都保存在同一目录下，使用一行 HOST COPY 指令就可以完成数据文件的拷贝。

【实例 25-46】备份数据文件。

```
SQL> host copy F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\*.* f:\user_offline_backup
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL01.CTL
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL02.CTL
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL03.CTL
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO01.LOG
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO02.LOG
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO03.LOG
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\TEMP01.DBF
```

```
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
已复制         12 个文件。
```

注意此时是在数据库关闭状态下执行的，使用 HOST 指令说明后面的指令是操作系统指令，在 UNIX 系统中使用 HOST CP 指令拷贝文件到指定目录。此时成功完成数据库的用户管理的脱机备份，然后启动数据库。

4. 启动数据库

【实例 25-47】启动数据库完成脱机备份。

```
SQL> startup
ORACLE 例程已经启动。

Total System Global Area  603979776 bytes
Fixed Size                  1250380 bytes
Variable Size             218106804 bytes
Database Buffers          377487360 bytes
Redo Buffers                7135232 bytes
数据库装载完毕。
数据库已经打开。
```



用户管理的脱机备份中，备份的数据文件只能恢复备份之前的数据，如果之后的数据库结构有变化，如创建了表空间、增加了数据文件等，控制文件最好重新备份。总之用户管理的冷备份使得所有的数据库文件定格在一个时，对于这个时刻之后的数据变化是无法恢复的。并且对于 7*25 小时工作的生产数据库而言，脱机冷备份往往是不可取的，需要采取热备份的方式。

25.5.2　从脱机备份中手工恢复

使用用户管理的脱机备份作为恢复数据源时，需要考虑数据库的归档模式，如果数据库处于归档模式，需要使用数据恢复将备份后的所有变化了的数据重写进数据文件中。如果数据库处于非归档模式，则只需要从最近的脱机备份中拷贝数据库的数据文件、联机重做日志文件和控制文件。下面分两种情况详细介绍从用户管理的脱机冷备份中手工恢复。

在恢复的数据库不处于归档模式，首先保证数据库处于关闭状态，然后将脱机备份文件拷贝到当前数据库的相应目录下，然后重启数据库，具体步骤如下。

- 关闭数据库，此时可以使用 SHUTDOWN ABORT 来关闭数据库。
- 从最近的脱机备份中拷贝数据库文件，包括数据文件、日志文件和控制文件。此时需要用户事先知道当前数据库的这些文件的位置。
- 重启数据，使用 STARTUP 指令。

上述的恢复方法可以恢复一个数据文件或整个数据库。当数据库处于归档模式时，就有所不同了，但它和从热备份执行恢复一样。

25.5.3 联机备份方法

联机备份是指在数据库不关闭的情况下实现备份，而用户管理更强调在这个过程中用户手工操作的过程，如设置备份模式、拷贝数据文件、备份重做日志文件和归档日志文件等，本节将讲解如何实现用户管理的联机备份，显然联机备份是在运行的数据库上实现备份操作，必须将要备份的表空间置于备份模式，这种模式的含义是告诉数据库该表空间中的数据文件正在备份，不能对它进行修改操作，但是可以读取，也不能再向该表空间写入数据，处于备份模式的表空间中的数据不再变化。

那么在备份期间变化的数据，或要写入备份表空间中的数据该如何处理呢？显然需要这个变化的一个备份，读者应该知道归档重做日志的作用吧，这里就使用归档重做日志记录在联机备份期间变化的数据，所以要使用 Oracle 数据库的联机（热）备份，数据库必须处于归档（ARCHIVELOG）模式，并且要求在备份期间产生归档日志。

1. 准备工作

在实现用户管理的联机热备份前必须做一些准备工作。

（1）将数据库设置为归档模式

设置数据库为归档模式，在 Oracle 9i 和 Oracle 11g 中方法是一样的，如果数据库打开，则必须首先安全地关闭数据库，启动数据库到 MOUNT 状态，如下所示。

【实例 25-48】重启数据库到 MOUNT 状态。

```
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL> startup mount
ORA-32004: obsolete and/or deprecated parameter(s) specified
ORACLE 例程已经启动。

Total System Global Area  603979776 bytes
Fixed Size                 1250380 bytes
Variable Size            234884020 bytes
Database Buffers          360710144 bytes
Redo Buffers               7135232 bytes
数据库装载完毕。
```

然后使用如实例 25-49 所示的方法将数据库置于归档模式。

【实例 25-49】设置数据库为归档模式。

```
SQL> alter database archivelog;

数据库已更改。
```

此时将数据库改变到 OPEN 状态，即在数据库处于 MOUNT 状态时打开数据库，此时可以使用 ARCHIVE LOG LIST 指令查看数据库的归档信息。

【实例 25-50】打开数据库。

```
SQL> alter database open;

数据库已更改。
```

【实例 25-51】查看数据库的归档信息。

```
SQL> archive log list
数据库日志模式            存档模式
自动存档             启用
存档终点            USE_DB_RECOVERY_FILE_DEST
最早的联机日志序列     184
下一个存档日志序列   186
当前日志序列           186
```

从上例的输出可以看出，当前的数据库日志模式处于归档模式，而存档终点是 USE_DB_RECOVERY_FILE_DEST，该值的含义是使用数据库快闪恢复区的数据库存储目录。

（2）设置归档日志存储参数

将参数 LOG_ARCHIVE_DEST_1 的值设置为存储归档日志的目录。如果使用了快闪恢复区作为归档日志文件的存储目录，也可以不设置该参数。将参数 LOG_ARCHIVE_START 的值设置为 TRUE。

2. 用户管理的数据文件联机热备份过程

此时用户可以选择备份的数据文件，可以是整个数据库，也可以是某个数据文件。如果只备份某个表空间的数据文件，可以首先将该表空间置于备份模式，如果要备份整个数据库，则需要将所有数据文件所在的表空间置于备份模式。下面将演示如何联机手工备份一个数据文件。首先需要确定数据文件的操作系统存储目录，以及数据文件对应的表空间，目的是把这些表空间设置为备份模式。

【实例 25-52】查找数据文件以及对应的表空间。

```
SQL> col file_name for a55
SQL> select file_name ,tablespace_name
  2* from dba data files

FILE_NAME                                               TABLESPACE_NAME
------------------------------------------------------- ---
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF        USERS
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF      SYSAUX
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF     UNDOTBS1
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF      SYSTEM
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF     EXAMPLE
```

此时，检查数据文件目录的目的是拷贝这些文件，而查看表空间的目的是设置这些表空间为备份模式。下面需要将所有表与数据文件相关的表空间置于备份模式，在 Oracle 9i 或以上版本中都可以使用 ALTER TABLESAPCE tbs_name BEGIN BACKUP 将该表空间置于备份模式。

【实例 25-53】将表空间设置为备份模式。

```
SQL> alter tablespace users begin backup ;

表空间已更改。
```

此时表空间就置于备份模式，为了拷贝所有的数据文件，需要使用上述命令将所有的数据文件相关的表空间设置为备份模式。在 Oracle 11g 及以上版本中，可以使用一条指令将整个数据库置于备份模式。

【实例 25-54】将整个数据库置于备份模式。

```
SQL> alter database begin backup ;

数据库已更改。
```

此时，整个数据库的所有表空间都置于备份模式，现在可以拷贝数据文件到备份目录了，如实例 25-55 所示。

【实例 25-55】拷贝数据文件到备份目录。

```
SQL> host copy F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF f:\onlinebackup
已复制         1 个文件。
```

只要重复使用 HOST COPY 指令将需要的数据文件拷贝到备份目录即可。最后需要结束数据库或表空间的备份模式，如将某个表空间退出备份模式，如实例 25-56 所示。

【实例 25-56】把表空间退出备份模式。

```
SQL> alter tablespace users end backup;

表空间已更改。
```

如果在备份数据库时，将整个数据库置于备份模式，则需要使用如下方式将数据库退出备份模式。

【实例 25-57】将数据库退出备份模式。

```
SQL> alter database begin backup ;

数据库已更改。
```

3. 备份归档日志文件

可以使用如下方式查看当前数据库的归档日志存储目录，从而拷贝在备份过程中产生的归档日志数据。

【实例 25-58】查看归档日志位置。

```
SQL> col name for a2
SQL> col name for a25
SQL> col value for a50
SQL> select name,value from v$parameter where name in
  2* ('log_archive_dest','log_archive_dest_1','db_recovery_file_dest')
```

```
NAME                      VALUE
------------------------- ------------------------------------------------
log_archive_dest
log_archive_dest_1
db_recovery_file_dest     F:\oracle\product\10.2.0/flash_recovery_area
```

然后到归档日志目录下备份联机备份过程中（开始备份模式和退出备份模式之间）产生的归档重做日志。

4. 备份控制文件

在备份了数据文件后应该对控制文件做备份，因为在备份数据文件后，整个数据库结构可能发生了变化，如新建了数据表空间、当前数据表空间增加了数据文件等。这些都会记录在控制文件中，而备份的数据文件信息记录在当前的控制文件中，如果在将来需要做数据库的介质恢复，就使用我们备份的数据文件，也需要使用此时备份的控制文件。备份控制文件的方式如下所示。

【实例 25-59】备份控制文件。

```
SQL> alter database backup controlfile to 'f:\onlinebackup\mycontrolfile.ctl';

数据库已更改。
```

下面通过操作系统指令查看是否在指定目录 f:\onlinebackup 下生成了控制文件的备份文件。

【实例 25-60】查看是否成功创建备份的控制文件。

```
F:\onlinebackup>dir
 驱动器 F 中的卷是 Oracle11g
 卷的序列号是 000B-1DED
F:\onlinebackup 的目录

009-08-30  19:29    <DIR>          .
009-08-30  19:29    <DIR>          ..
009-08-30  18:40         7,061,504 CONTROL01.CTL
………..
009-08-30  19:29         7,061,504 MYCONTROLFILE.CTL
…………
009-08-30  18:40       104,865,792 USERS01.DBF
              13 个文件  2,055,980,544 字节
               2 个目录 70,464,360,448 可用字节
```

从输出可以看出在操作系统的 f:\onlinebackup 目录下成功备份了控制文件，文件名为 MYCONTROLFILE.CTL。

25.5.4　从联机备份中手工恢复

当数据库处于非归档模式时，可以通过联机备份的数据实现数据恢复，因为联机备份时，处于备份状态的表空间或整个数据库是无法写入数据的，虽然可以访问和使用 DML 操作，但是更新的数据只保留在重做日志文件中，所以采用联机备份实现恢复时，需要 RECOVER 数据，即将用

户提交的、记录在重做日志中的数据重新写到数据文件中。

下面通过一个具体实例说明在当前数据库处于非归档模式下时，如何实现数据恢复。以恢复一个数据文件为例，假设表空间 USERS 中的数据文件 USERS01.DBF 损坏，或者其中的部分重要的表被删除。具体步骤如下。

将数据文件 USERS01.DBF 置于脱机状态，之前需要使用数据字典 DBA_DATA_FILES 来查看表空间 USERS 中的数据文件在操作系统上的目录。下面将设置数据文件为脱机状态。

【实例 25-61】将数据文件设置为脱机状态。

```
SQL> alter database datafile 'F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\
USERS01.DBF' offline;

数据库已更改。
```

下面查看数据文件 USERS01.DBF 的状态信息，如下所示。

【实例 25-62】查看脱机数据文件 USERS01.DBF 的状态信息。

```
SQL> select file_name,status,online_status
  2  from dba_data_files
  3  where tablespace name ='USERS';

FILE_NAME                                              STATUS    ONLINE_
---------------------------------------------------------------------------
F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF  AVAILABLE RECOVER
```

上述输出说明文件名 FILE_NAME 为需要恢复的数据文件，而 ONLINE_STATUS 的值为 RECOVER，说明该数据文件需要介质恢复，说明数据文件中的 SCN 与控制文件中的 SCN 不一致，需要使用重做日志文件中的数据恢复用户提交的数据，一旦将处于非归档模式的数据库中的数据文件设置为脱机状态，则该数据文件的 ONLINE_STATUS 值自动为 RECOVER，指示需要介质恢复。

此时需要将脱机备份的数据文件拷贝到当前数据库中该文件的目录下，然后实现介质恢复。如何实现数据文件的介质恢复呢？答案是使用 RECOVER DATAFILE 指令，如实例 25-63 所示。

【实例 25-63】实现数据文件的介质恢复。

```
SQL> recover datafile 'F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF';
完成介质恢复。
```

虽然实现了介质恢复，但是此时的数据文件仍然是不可访问的，如果试图访问该数据文件中涉及的表，则提示如下错误。

【实例 25-64】访问脱机文件涉及的表数据。

```
SQL> select *
  2  from scott.dept;
from scott.dept
          *
第 2 行出现错误:
ORA-00376: 此时无法读取文件 4
ORA-01110: 数据文件 4: 'F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF'
```

显然，错误很明显，由于数据文件没有在线，使得 Oracle 无法读取该文件，下面通过实例 25-65 将数据文件 USERS01.DBF 设置为在线状态，此时用户可以访问该数据文件了。

【实例 25-65】将数据文件设置为在线状态。

```
SQL> alter database datafile 'F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF'
online;

数据库已更改。
```

此时，已成功通过脱机备份的数据文件为处于归档模式数据库恢复了数据文件 USERS01.DBF。


说　明

如果读者是实验数据库，而且有脱机备份的所有数据文件，可以先关闭数据库，然后删除 USERS 表空间中的 USERS01.DBF 文件来模拟数据文件丢失。这种情况下，数据库无法正常启动，但是只要使用将脱机的备份数据文件拷贝到当前 USERS01.DBF 数据文件的目录下，然后使用 RECOVER DATAFILE 指令实现数据文件的介质恢复，就可以正常打开数据库了。

25.6　本章小结

本章介绍了数据库备份的各种方法以及使用工具。其中 EXP/IMP 是 Oracle 较古老的备份恢复数据库工具，它实现了数据库的逻辑备份，而数据泵技术是 Oracle 推荐的代替 EXP/IMP 的数据库备份和恢复工具，使用数据泵技术具有备份和恢复数据库更快、重启失败作业、实现交互模式、支持网络操作等优势。

第 26 章

RMAN备份与恢复数据库

RMAN 是 Oracle 提供的使用程序 Recovery Manager，即恢复管理器，使用 RMAN 可以轻松实现数据库的所有备份任务，它可以通过命令行方式操作，也可以通过 OEM 的 Database Control 界面实现，本章将详细介绍 RMAN 技术的每一个细节。

26.1 RMAN 概述

RMAN 在数据库服务器的帮助下实现数据库文件、控制文件、数据库文件和控制文件的映像副本，以及归档日志文件、数据库服务器参数文件的备份。RMAN 也允许使用脚本文件实现数据的备份与恢复，而且这些脚本保存在数据库内，且不需要编写基于 OS 的脚本文件。RMAN 备份的文件自动保存在一个系统指定的目录下，文件的名称也由 RMAN 自己维护，实现数据恢复操作时，恢复指令简洁，RMAN 自动寻找需要的文件实现数据恢复。减少了在传统的导出导入程序中人为错误的发生。

26.1.1 RMAN 的特点

如果读者使用过 EXP/IMP 以及 EXPDP/IMPDP 工具，应该很好理解使用 RMAN 带来的好处，Oracle 每次技术的演进都是使得其功能更强大、操作更简单，更加满足生产数据库的要求。相对“古老”的备份技术而言，使用 RMAN 的优点如下所示。

- 支持增量备份：在传统的备份工具中如 EXP 或 EXPDP，只能实现一个完整备份而不能增量备份，RMAN 采用被备份级别实现增量备份，在一个完整备份的基础上，采用增量备份，和传统备份方式相比，可以减少备份的数据量。
- 自动管理备份文件：RMAN 备份的数据是 RMAN 自动管理的，包括文件名字、备份文件存储目录，以及识别最近的备份文件、搜索恢复时需要的表空间、模式或数据文件等备份文件。
- 自动化备份与恢复：在备份和恢复操作时，使用简单的指令就可以实现备份与恢复，且执

行过程完全由 RMAN 自己维护。

- 不产生重做信息：与用户管理的联机备份不同，使用 RMAN 的联机备份不产生重做信息。
- 恢复目录：RMAN 的自动化备份与恢复功能应该归功于恢复目录的使用，RMAN 直接在其中保存了备份和恢复脚本。
- 支持映像拷贝：使用 RMAN 也可以实现映像拷贝，映像是以操作系统上的文件格式存在，这种拷贝方式类似于用户管理的脱机备份方式。
- 新块的比较特性：这是 RMAN 支持增量备份的基础，这种特性使得在备份时，跳过数据文件中从未使用过的数据块的备份，备份数据量的减少直接导致了备份存储空间需求和备份时间的减少。
- 备份的数据文件压缩处理：RMAN 提供一个参数，说明是否对备份文件进行压缩，压缩的备份文件以二进制文件格式存在，可以减少备份文件的存储空间。
- 备份文件有效性检查功能：这种功能验证备份的文件是否可用，在恢复前往往需要验证备份文件的有效性。

26.1.2　RMAN 的系统结构组成

Oracle 的 RMAN 工具使用会话建立客户端到数据库服务器的连接，用户首先需要启动 RMAN 可执行程序，然后建立客户端与服务器端的会话连接，用户通过 RMAN 的客户端进行 RMAN 操作，执行备份与恢复指令，这些指令在服务器端的服务器进程中执行，而服务器进程完成实际的磁盘读写操作。下面，为了完成数据库的备份与恢复操作，将详细介绍 RMAN 的系统结构组成。

- RMAN 可执行程序：它是一个客户端工具，用来启动与数据库服务器的连接，从而实现备份与恢复的各种操作。
- RMAN 客户端：一旦建立了与数据库服务器的会话连接，RMAN 可执行程序就创建一个客户端，通过客户端完成与数据库服务器之间的通信，完成各种备份与恢复操作的指令。RMAN 客户端可以通过 Oracle Net 连接到可访问的任何主机上。
- 服务器进程：在 RMAN 建立了与数据库服务器的会话连接后，在数据库服务器端启动一个后台进程，它执行 RMAN 客户端发出的各种数据恢复与备份指令，并完成实际的磁盘或磁带设备的读写任务。
- RMAN 信息库：RMAN 信息库记录了 RMAN 的一些信息，如备份的数据文件及副本的目录、归档的重做日志备份文件和副本、表空间和数据文件、备份或恢复的脚本、RMAN 的配置信息。默认使用数据库服务器的控制文件记录这些信息，读者可以通过转储的控制文件发现这些信息，如使用 ALTER DATABASE BACKUP CONTROL FILE TO TRACE。
- 恢复目录：记录 RMAN 信息库的信息。但是恢复目录需要事先配置，信息库既可以存储在数据库的控制文件中，也可以存储在恢复目录中。在 Oracle 中默认先将 RMAN 信息库写入控制文件，如果存在恢复目录则需要继续写到恢复目录。使用控制文件的不足是控制文件中记录 RMAN 信息库的空间有限，当空间不足时可能被覆盖掉，所以 Oracle 建议创建单独的恢复目录，这样也可以更好地发挥 RMAN 提供的新特性。

图 26-1 给出了 RMAN 的系统结构图，其实也可以理解为一个备份或恢复过程的信息流示意图，

RMAN 可执行程序启动并建立与数据库服务器的会话连接，客户端发出备份指令，而数据库服务器端的服务器后台进程执行指令完成磁盘读写操作，并将备份信息记录在 RMAN 信息库中，RMAN 信息库可以保存在数据库服务器端的控制文件中，如果使用恢复目录，RMAN 信息库同样会自动保存在恢复目录中。

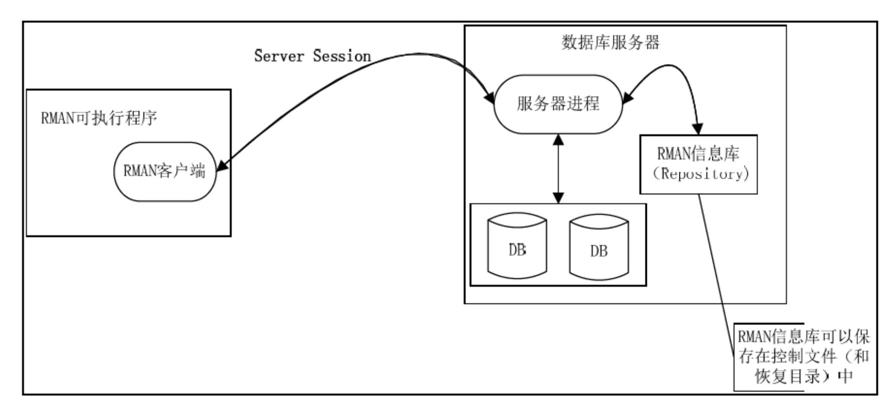


图 26-1　RMAN 的系统结构组成

26.1.3　RMAN 的快闪恢复区

快闪恢复区是存储、备份、恢复数据文件以及相关信息的存储区。快闪恢复区保存了每个数据文件的备份、增量备份、控制文件备份以及归档重做日志备份，Oracle 也允许在快闪恢复区中保存联机重做日志的冗余副本以及当前控制文件的冗余副本，Oracle 中闪回特性中的闪回日志也保存在快闪恢复区中。

在使用 RMAN 实现数据库的备份与恢复时，配置的快闪恢复区就是 RMAN 存储所有与备份相关的文件存储区，而此时的文件名不需要用户干预，Oracle 使用 OMF 创建备份文件的文件名。

使用快闪恢复区的优点是：实现了备份文件的自动管理，使得备份与恢复数据库更简单（指令更简洁），并且可以集中管理磁盘空间。

恢复区的空间足够大，以容纳备份的数据。那么如何管理快闪恢复区呢？在 Oracle 中快闪恢复区由两个初始化参数设置，另一个是 DB_RECOVERY_FILE_DEST_SIZE，该参数用于设置快闪恢复区的最大容量，一个是 DB_RECOVERY_FILE_DESC，该参数用于设置快闪恢复区在操作系统磁盘空间上的位置，可以通过两种方式来设置快闪恢复区的参数：一种方法是通过在初始化参数文件 init.ora 中设置这两个参数；另一种方法是通过数据库指令 ALTER SYSTEM 在运行的数据库上动态地设置。下面演示如何动态设置快闪恢复区的参数。

首先查看当前数据库的快闪恢复区参数，使用 SHOW PARAMETER 指令实现。

【实例 26-1】查看快闪恢复区的参数信息。

```
SQL> show parameter db_recovery

NAME                                TYPE        VALUE
------------------------------------ ----------------------------------
db_recovery_file_dest                 string         F:\oracle\product\10.2.0/flash
                                                   _recovery_area
```

```
db_recovery_file_dest_size           big integer     2G
```

从中可以看到快闪恢复区在磁盘上的目录和快闪恢复区的空间大小，备份的整个数据库以及控制文件都保存在该快闪恢复区中，该区域中的文件由 Oracle 自己维护，一旦需要恢复数据库时，只需要使用简单地指令就可以恢复数据库，RMAN 工具会自动寻找存储在快闪恢复区中的备份文件完成恢复。

快闪恢复区的参数可以动态更改，如可以在数据库运行期间改变快闪恢复区的大小，以及改变快闪恢复区在磁盘上的存储目录。

【实例 26-2】修改快闪恢复区的参数。

```
SQL> alter system set
  2  db_recovery_file_dest_size=2g;

系统已更改。

SQL> alter system set
  2  db_recovery_file_dest = 'f:\flashrecovery_area'

系统已更改。
```

快闪恢复区的参数除了在运行库上动态更改，或者在 init.ora 文件或 SPFILE 文件中设置，也可以使用 OEM 工具的 DATABASE CONTROL 配置快闪恢复区。

为了以后演示方便，仍将快闪恢复区的目录设置为其默认目录，大小仍为 2G。

如果不需要快闪恢复区可以将参数 DB_RECOVERY_FILE_DEST 的值设置为空格，使得快闪恢复区不存在存储目录。

当使用了快闪恢复区后，可以通过数据字典 v$recovery_file_dest 来查看快闪恢复区的空间使用情况以及文件数量。

【实例 26-3】查看快闪恢复区的位置以及空间使用信息。

```
SQL> col name for a30
SQL> set line 100
SQL> select name,space_limit,space_used,number_of_files
  2* from v$recovery file dest

NAME                     SPACE_LIMIT SPACE_USED NUMBER_OF_FILES
-------------------------------------- ------------------- ----
F:\oracle\product\10.2.0/flash    2147483648   793569280              7
_recovery_area
```

上述输出说明当前数据库的快闪恢复区的空间 SPACE_LIMIT 为 2G，已经使用了 SPACE_USED 为 756M，当前恢复区中的文件数为 7。NAME 的值说明快闪恢复区的操作系统目录，该目录为 F:\oracle\product\10.2.0\flash_recovery_area。

那么如果快闪恢复区的空间不足该如何处理呢？有三种方法，一是增加恢复区磁盘空间，但这受当前磁盘空间的限制，二是删除没用的备份文件或将备份文件拷贝到磁带设备，三是删除当前的恢复区，重新设置新的快闪恢复区。

● 增加磁盘空间：可以使用 ALTER SYSTEM 指令动态设置快闪恢复区的空间大小，如实例 26-4 所示。
● 使用 CROSSCHECK 和 DELETE EXPIRED 指令删除不需要的文件。使用 RMAN 的 BACKUP RECOVERY AREA 指令将恢复区中的文件拷贝到磁带中。
● 删除当前的快闪恢复区，并重新设置："SQL> alter system set db_recovery_file_dest ='f:\newflasharea'"。

【实例 26-4】重新设置快闪恢复区的空间大小。

```
SQL> alter system set
  2  db_recovery_file_dest_size=4g;

系统已更改。
```



当向快闪恢复区添加新文件时，Oracle 会自动更新文件列表，发现符合删除条件的备份文件，这些文件包括不符合保留策略的文件，拷贝到磁带的过渡文件，而重做日志文件和控制文件任何时候都不会被删除。

26.1.4　RMAN 到数据库的连接

本节将讲解如何使用 RMAN 建立到数据库服务器的连接。通过实例说明连接到数据库服务器。

【实例 26-5】使用数据库用户名和密码登录 RMAN。

```
D:\>rman

恢复管理器: Release 11.1.0.6.0 - Production on 星期一 8月 31 22:10:05 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

RMAN> connect target system/oracle@orcl

连接到目标数据库: ORCL (DBID=1219822601)
```

上例说明首先在操作系统环境下输入 RMAN 指令，启动 RMAN 可执行程序，而后通过 connect target 指令使用数据库用户名和密码建立与数据库服务器的会话连接。

【实例 26-6】使用操作系统认证连接到 RMAN。

```
D:\>rman target /

恢复管理器: Release 11.1.0.6.0 - Production on 星期一 8月 31 22:12:41 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

已连接到目标数据库: LEEJIA (DBID=587536714)
```

26.2　使用 RMAN 实现备份

使用 RMAN 进行更多类型的备份前，有必要说明几个 RMAN 概念，这些概念也多次出现在备份输出过程中，如下所示。

- 备份集：备份集是一个逻辑数据集合，由一个或多个 RMAN 的备份片组成，备份片是 RMAN 格式的操作系统文件，包含一个数据文件、一个控制文件或者归档日志文件。默认情况下，在执行 RMAN 的备份时，将产生备份文件的备份集，备份集只用 RMAN 识别，所以在恢复时必须使用 RMAN 来访问备份集实现恢复。
- 通道：RMAN 是通过与数据库服务器的会话建立连接，通道代表这个连接，它指定了备份或恢复数据库的备份集所在的设备，如磁盘或磁带。
- 映像拷贝：映像拷贝是数据库文件的操作系统文件的一个备份，就如使用操作系统的 COPY 指令备份的文件一样。使用 RMAN 将默认创建备份集，它是数据集的一个逻辑数据结构，也可以设置备份类型为 COPY，使得使用 RMAN 的任何备份不产生备份集，而产生映像拷贝，如下所示：也可以使用 BACKUP AS COPY 指令实现备份数据的映像拷贝，如下所示为实现整个数据库的映像拷贝。

```
RMAN> CONFIGURE DEVICE TYPE DISK BACKUP TYPE TO COPY
```

【实例 26-7】映像拷贝整个数据库。

```
RMAN> BACKUP AS COPY DATABASE
```

【实例 26-8】映像拷贝单个表空间。

```
RMAN> BACKUP AS COPY TABLESPACE USERS
```

【实例 26-9】映像拷贝整个数据库的一个数据文件。

```
RMAN> BACKUP AS COPY DATAFILE 3
```

参数 DATAFILE 后的数值表示数据文件 ID，它与数据字典 DBA_DATA_FILES 中的 FILE_ID 参数一致。接下来分析一下 RMAN 的配置参数，首先登录 RMAN 之后，使用 SHOW ALL 指令显示当前的所有 RMAN 参数。

【实例 26-10】查看 RMAN 的配置参数。

```
RMAN> show all;

RMAN 配置参数为:
CONFIGURE RETENTION POLICY TO REDUNDANCY 1; # default
CONFIGURE BACKUP OPTIMIZATION OFF; # default
CONFIGURE DEFAULT DEVICE TYPE TO DISK; # default
CONFIGURE CONTROLFILE AUTOBACKUP OFF; # default
CONFIGURE CONTROLFILE AUTOBACKUP FORMAT FOR DEVICE TYPE DISK TO '%F'; # default
CONFIGURE DEVICE TYPE DISK PARALLELISM 1 BACKUP TYPE TO BACKUPSET; # default
CONFIGURE DATAFILE BACKUP COPIES FOR DEVICE TYPE DISK TO 1; # default
CONFIGURE ARCHIVELOG BACKUP COPIES FOR DEVICE TYPE DISK TO 1; # default
```

```
CONFIGURE MAXSETSIZE TO UNLIMITED; # default
CONFIGURE ENCRYPTION FOR DATABASE OFF; # default
CONFIGURE ENCRYPTION ALGORITHM 'AES128'; # default
CONFIGURE ARCHIVELOG DELETION POLICY TO NONE; # default
CONFIGURE SNAPSHOT CONTROLFILE NAME TO 'F:\ORACLE\PRODUCT\10.2.0\DB_1\
DATABASE\SNCFORCL.ORA'; # default
```

可以根据需要更改上述中的参数，先解释部分 RMAN 参数的含义，然后说明如何设置该参数。

CONFIGURE RETENTION POLICY TO REDUNDANCY 1：该参数说明保留备份的副本数量，如果每天都备份一个数据文件，上述参数 1 说明只保留一个该数据文件的副本。

CONFIGURE DEFAULT DEVICE TYPE TO DISK：该配置参数说明备份的数据文件默认备份到数据库服务器的磁盘上，该参数可以更改为备份到磁带上，如下例所示。

【实例 26-11】更改 RMAN 的备份设备类型为磁带。

```
RMAN> configure default device type to sbt;

新的 RMAN 配置参数：
CONFIGURE DEFAULT DEVICE TYPE TO 'SBT_TAPE';
已成功存储新的 RMAN 配置参数
释放的通道：ORA_DISK_1
```

这里仅仅是为了说明设备类型的更改方式，读者最好使用如下方式将设备类型恢复为磁盘：

```
RMAN> configure default device type to disk;
```

CONFIGURE BACKUP OPTIMIZATION OFF：配置备份优化，模式不使用备份优化，使用备份优化的作用是如果已经备份了某个文件的相同版本，则不会再备份该文件。可以使用如下方式打开备份优化：

```
RMAN> CONFIGURE BACKUP OPTIMIZATION ON;

新的 RMAN 配置参数：
CONFIGURE BACKUP OPTIMIZATION ON;
已成功存储新的 RMAN 配置参数
```

CONFIGURE CONTROLFILE AUTOBACKUP OFF：配置模式不启动控制文件的自动备份，更改方式就是将 OFF 设置为 ON，修改指令如下所示：

```
RMAN> CONFIGURE CONTROLFILE AUTOBACKUP ON;
```

CONFIGURE DEVICE TYPE DISK PARALLELISM 1 BACKUP TYPE TO BACKUPSET：该参数说明 RMAN 在备份和恢复中运行的通道数量，在执行备份或恢复时，通道数量越多，则任务执行时间越短。修改并行数的指令如下所示：

```
RMAN> CONFIGURE DEVICE TYPE DISK PARALLELISM 3;
```

26.2.1 使用 RMAN 实现脱机备份

下面通过实例说明如何实现 RMAN 的脱机备份，要实现脱机备份首先需要使用 RMAN 登录到数据库服务器，关闭数据库然后启动数据库到 MOUNT 状态，再执行 BACKUP DATABASE 指

令备份整个数据库，具体步骤如下所示。

01 使用数据库用户名和密码登录 RMAN:

```
D:\>rman target system/oracle@orcl
恢复管理器: Release 11.1.0.6.0 - Production on 星期六 8 月 29 16:48:49 2009
Copyright (c) 1982, 2007, Oracle.  All rights reserved.
连接到目标数据库: ORCL (DBID=1219822601)
RMAN>
```

02 在 RMAN 执行程序中，通过客户端指令关闭数据库，然后从 RMAN 加载数据到 MOUNT 状态:

```
RMAN> shutdown immediate
使用目标数据库控制文件替代恢复目录
数据库已关闭
数据库已卸载
Oracle 实例已关闭
RMAN> startup mount
已连接到目标数据库 (未启动)
Oracle 实例已启动
数据库已装载
系统全局区域总计      603979776 字节

Fixed Size                    1260380 字节
Variable Size               226495412 字节
Database Buffers            369098752 字节
Redo Buffers                  7135232 字节
```

03 使用 BAKCUP DATABASE 备份指令备份整个数据库，如没有配置快闪恢复区，则需要使用 FORMAT 参数说明要备份的全库的备份集放在哪个目录下。

```
RMAN> backup database;

启动 backup 于 01-9 月 -09
使用通道 ORA_DISK_1
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
输入数据文件 fno=00002
name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
输入数据文件 fno=00001
name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
输入数据文件 fno=00003
name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
输入数据文件 fno=00005
name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
输入数据文件 fno=00004
name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
通道 ORA_DISK_1: 正在启动段 1 于 01-9 月 -09
通道 ORA_DISK_1: 已完成段 1 于 01-9 月 -09
段句柄
```

```
=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2009_09_
01\O1_MF_NNNDF_TAG20090901T223404_59TD6X2G_.B
KP 标记=TAG20090901T223404 注释=NONE
通道 ORA_DISK_1: 备份集已完成, 经过时间:00:02:46
完成 backup 于 01-9月 -09

启动 Control File and SPFILE Autobackup 于 01-9月 -09
段
handle=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\AUTOBACKUP\2
009 09 01\O1 MF S 696465171 59TDD3LK .BKP commen
t=NONE
完成 Control File and SPFILE Autobackup 于 01-9月 -09

RMAN>
```

04 当备份整个数据库时，RMAN 将自动备份控制文件和服务器参数文件，其实这取决于 RMAN 的 CONFIGURE CONTROLFILE AUTOBACKUP，可以设置该参数值为 ON，使得在使用 RMAN 执行任何备份指令时，自动备份控制文件和 SPFILE 文件。参数配置如下所示：

```
RMAN> configure controlfile autobackup on;
```

05 在上例中，就启用了控制文件的自动备份，这样在使用 RMAN 执行任何数据备份时都自动备份控制文件。在备份整个数据库时，如启用了快闪恢复区，则使用简单的 BAKCUP DATABASE 备份指令。

06 最后重启数据库。

【实例 26-12】重启数据库。

```
RMAN> alter database open;

数据库已打开
```

此时，使用 RMAN 完成了整个数据库的脱机备份。

26.2.2 使用 RMAN 实现控制文件备份

RMAN 可以读单独备份控制文件，如果没有启用快闪恢复区，则使用 FORMAT 参数指定控制文件的备份目录，如果启用了快闪恢复区，RMAN 会自动将控制文件拷贝到快闪恢复区的备份集中（BACKUPSET 目录下），下面通过两个实例演示如何使用备份控制文件的指令和整个备份过程。

【实例 26-13】在没有启用快闪恢复区时备份控制文件。

```
RMAN> backup current controlfile format
2> 'f:\pump\backup_ctl_%u.dbf';
启动 backup 于 29-8月 -09
使用通道 ORA_DISK_1
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
备份集中包括当前控制文件
```

```
通道 ORA_DISK_1: 正在启动段 1 于 29-8 月 -09
通道 ORA_DISK_1: 已完成段 1 于 29-8 月 -09
段句柄=F:\PUMP\BACKUP_CTL_0CKNTU1G.DBF 标记=TAG20090829T171527 注释
=NONE
通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:02
完成 backup 于 29-8 月 -09
```

在本章的备份实例中多次使用替换变量“%U”，它的作用是产生唯一的备份文件名。因为没有使用快闪恢复区，所以在执行控制文件恢复时，DBA 必须知道备份目录，显然这增加了 DBA 的工作负担。下面是使用快闪恢复区时的备份控制文件方式。

【实例 26-14】在启用快闪恢复区时备份控制文件。

```
RMAN> backup current controlfile
2> ;

启动 backup 于 29-8 月 -09
使用通道 ORA_DISK_1
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
备份集中包括当前控制文件
通道 ORA_DISK_1: 正在启动段 1 于 29-8 月 -09
通道 ORA_DISK_1: 已完成段 1 于 29-8 月 -09
段句柄
=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2009_08_
29\O1_MF_NCNNF_TAG20090829T171726_59KWK66T_.B
KP 标记=TAG20090829T171726 注释=NONE
通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:02
完成 backup 于 29-8 月 -09
```

使用快闪恢复区的好处就是 Oracle 自动管理文件的备份目录，DBA 也不需要记住这个目录，在恢复时同样不需要记住该目录，RMAN 将使用 RMAN 信息库记录的信息找到备份的文件集。

26.2.3　使用 RMAN 实现联机备份

在使用 RMAN 进行联机备份前，必须设置快闪恢复区，将 DB_RECOVERY_FILE_DEST 参数指定的目录作为归档重做日志备份的默认位置，并且将快闪恢复区的尺寸设置的足够大，然后需要将 LOG_ARCHIVE_START 参数的值设置为 TRUE。

在进行联机备份前都要求将数据库置于归档模式，因为处于联机备份的数据库中要备份的所有数据文件头中的 SCN 被锁定，但是此时在数据库中的数据文件的表依然可以被访问，并执行 DML 操作，但是这些修改的数据不能写入数据文件，Oracle 的重做日志进程将这些变化的数据全部写到重做日志文件，如果备份的时间很长，而且在这期间产生了大量的变化数据，重做日志会切换，从而将这些变化的数据写到归档日志文件中。

显然，RMAN 的联机备份使得数据库可以继续运行，而且通过 RMAN 可以备份整个数据库、一个表空间或者一个数据文件，可以灵活选择备份的粒度，对于超大型数据库如果备份整个数据库则是相当耗时的，而生产数据库中往往只需要备份某个重要的表空间或数据文件。 联机备份时启用归档模式，不会丢失数据更新。在介质故障时，可以实现数据库的全恢复。但是请注意，必须小

心保存或备份归档日志，因为一旦它丢失或损坏就无法实现数据库的完全恢复。

【实例 26-15】将参数 LOG_ARCHIVE_START 设置为 TRUE。

```
SQL> alter system set log_archive_start = true scope =spfile;

系统已更改。
```

最后需要将数据库置为归档模式，其操作步骤是先关闭数据库再启动数据库到 MOUNT 状态，然后使用 ALTER DATABASE ARCHIVELOG 将数据库设置为归档模式。打开数据库就可以进行 RMAN 联机热备份了。

【实例 26-16】将数据库设置为归档模式。

```
SQL> shutdown immediate;
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL>
SQL> startup mount;
ORA-32004: obsolete and/or deprecated parameter(s) specified
ORACLE 例程已经启动。

Total System Global Area  603979776 bytes
Fixed Size                  1260380 bytes
Variable Size             263272628 bytes
Database Buffers          352321536 bytes
Redo Buffers                7135232 bytes
数据库装载完毕。
SQL> alter database archivelog;

数据库已更改。
```

为了验证修改结果，最好使用下例所示查看当前数据库的归档模式。

【实例 26-17】查看数据库的归档模式。

```
SQL> archive log list;
数据库日志模式            存档模式
自动存档             启用
存档终点            USE_DB_RECOVERY_FILE_DEST
最早的联机日志序列     187
下一个存档日志序列   189
当前日志序列           189
```

下面依次通过实例说明如何使用 RMAN 联机备份整个数据库、备份一个表空间、备份一个数据文件以及备份当前的控制文件。

使用 BACKUP DATABASE 联机备份整个数据库。

【实例 26-18】使用 RMAN 备份整个数据库。

```
RMAN> backup database;
```

```
启动 backup 于 01-9 月 -09
使用目标数据库控制文件替代恢复目录
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=141 devtype=DISK
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
输入数据文件 fno=00002 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
输入数据文件 fno=00001 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
输入数据文件 fno=00003 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
输入数据文件 fno=00005 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
输入数据文件 fno=00004 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
............
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
备份集中包括当前控制文件
在备份集中包含当前的 SPFILE
通道 ORA_DISK_1: 正在启动段 1 于 01-9 月 -09
通道 ORA_DISK_1: 已完成段 1 于 01-9 月 -09
段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\
ORCL\BACKUPSET\2009_09_01\O1_MF_NCSNF_TAG20090901T012634_59R229XS_.B
P 标记=TAG20090901T012634 注释=NONE
通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:03
完成 backup 于 01-9 月 -09 完成
```

在备份整个数据库时，其实就是备份了数据文件，其中包含了当前的控制文件和参数文件。而重做日志文件或归档日志文件不是联机状态数据库全备份的内容，所以使用联机热备份的数据库在数据恢复时需要 recover 数据库，即将联机备份开始到故障点之间的所有提交的数据重新写入数据文件。

【实例 26-19】使用 RMAN 备份表空间。

```
RMAN> backup tablespace sysaux;

启动 backup 于 01-9 月 -09
使用目标数据库控制文件替代恢复目录
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=142 devtype=DISK
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
输入数据文件 fno=00003 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
通道 ORA_DISK_1: 正在启动段 1 于 01-9 月 -09
通道 ORA_DISK_1: 已完成段 1 于 01-9 月 -09
段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\
2009_09_01\O1_MF_NNNDF_TAG20090901T010402_59R0N2K2_.B
KP 标记=TAG20090901T010402 注释=NONE
通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:26
完成 backup 于 01-9 月 -09
```

上述实例中备份了表空间 SYSAUX，备份的文件类型为备份集，在第一次使用 RMAN 备份生

成备份集时，在 RMAN 的快闪恢复区中会自动创建一个目录 BACKUPSET，又会根据日期创建新的目录，将一天中的备份集放在一个根据日期创建的目录下。

在备份一个数据文件前，先查看当前数据库中的所有数据文件，以便于选择需要备份的数据文件。

【实例 26-20】查看当前数据库的所有数据文件。

```
SQL> col file name for a55
SQL> set line 100
SQL> select file_id,file_name,tablespace_name
  2* from dba data files

   FILE_ID FILE_NAME                                              TABLESPACE_NAME
---------- ------------------------------------------------------- ------
         4 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF       USERS
         3 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF      SYSAUX
         2 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF     UNDOTBS1
         1 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF      SYSTEM
         5 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF     EXAMPLE
```

在备份数据文件时，可以使用 FILE_ID 或者文件名来标识数据文件，如备份表空间 SYSAUX 中的数据文件，它的 FILE_ID 为 3。

【实例 26-21】使用 RMAN 备份数据文件。

```
RMAN> backup datafile 3;

启动 backup 于 01-9 月 -09
使用通道 ORA_DISK_1
通道 ORA_DISK_1: 启动全部数据文件备份集
通道 ORA_DISK_1: 正在指定备份集中的数据文件
输入数据文件 fno=00003 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
通道 ORA_DISK_1: 正在启动段 1 于 01-9 月 -09
通道 ORA_DISK_1: 已完成段 1 于 01-9 月 -09
段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\
2009_09_01\O1_MF_NNNDF_TAG20090901T010753_59R0V9XS_.B
KP 标记=TAG20090901T010753 注释=NONE
通道 ORA_DISK_1: 备份集已完成，经过时间:00:00:26
完成 backup 于 01-9 月 -09
```

虽然在备份整个数据库时，RMAN 自动备份了控制文件，但是经常联机备份控制文件是好习惯，一旦数据库结构发生了变化，如新建了表空间、添加或删除了数据文件等。

【实例 26-22】使用 RMAN 备份控制文件。

```
RMAN> backup current controlfile;

启动 backup 于 01-9 月 -09
使用目标数据库控制文件替代恢复目录
分配的通道: ORA_DISK_1
```

```
    通道 ORA_DISK_1: sid=139 devtype=DISK
    通道 ORA_DISK_1: 启动全部数据文件备份集
    通道 ORA_DISK_1: 正在指定备份集中的数据文件
    备份集中包括当前控制文件
    通道 ORA_DISK_1: 正在启动段 1 于 01-9月 -09
    通道 ORA_DISK_1: 已完成段 1 于 01-9月 -09
    段   句   柄   =F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\
2009_09_01\O1_MF_NCNNF_TAG20090901T010441_59R0OBJM_.B
    KP 标记=TAG20090901T010441 注释=NONE
    通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:04
    完成 backup 于 01-9月 -09
```

26.2.4　使用 RMAN 实现增量备份

在使用 BACKUP DATABASE 时，都是全库备份，显然每次这样的备份很耗费时间，也占用磁盘空间，而 RMAN 的增量备份具有很多优势，它只备份自上次全备份以来变化的数据。显然增量备份比全库备份要块，因为增量备份的数据量明显减少（相对于全库备份而言）。

这里需要解释两个级别的增量备份，一个是级别 0 的增量备份和级别 1 的增量备份，其中级别 0 的增量备份与全库备份相同。而级别 1 备份执行的是差异备份，即对级别 0 备份后变化的数据做备份。显然级别 0 备份是级别 1 备份的数据基础。

【实例 26-23】使用 RMAN 实现增量备份的级别 0 备份。

```
    RMAN> backup incremental level 0 database;

    启动 backup 于 01-9月 -09
    使用通道 ORA_DISK_1
    通道 ORA_DISK_1: 启动增量级别 0 数据文件备份集
    通道 ORA_DISK_1: 正在指定备份集中的数据文件
    输入数据文件 fno=00002 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
    输入数据文件 fno=00001 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
    输入数据文件 fno=00003 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
    输入数据文件 fno=00005 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
    输入数据文件 fno=00004 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
    通道 ORA_DISK_1: 正在启动段 1 于 01-9月 -09
    通道 ORA_DISK_1: 已完成段 1 于 01-9月 -09
    段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\
    BACKUPSET\ 2009_09_01\O1_MF_NNND0_ TAG20090901T171114_59SS9MDP_.B
    KP 标记=TAG20090901T171114 注释=NONE
    通道 ORA_DISK_1: 备份集已完成, 经过时间:00:02:56
    通道 ORA_DISK_1: 启动增量级别 0 数据文件备份集
    通道 ORA_DISK_1: 正在指定备份集中的数据文件
    备份集中包括当前控制文件
    在备份集中包含当前的 SPFILE
    通道 ORA_DISK_1: 正在启动段 1 于 01-9月 -09
    通道 ORA_DISK_1: 已完成段 1 于 01-9月 -09
    段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\
    2009_09_01\O1_MF_NCSN0_ TAG20090901T171114_59SSH4CQ_.B
    KP 标记=TAG20090901T171114 注释=NONE
```

```
    通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:03
    完成 backup 于 01-9月 -09
```

上例的全库备份集是增量备份的数据基础，在使用增量备份的级别 1 的第一次备份时，将变化的数据记录到增量备份集。而当第二次使用级别 1 增量备份时，只备份自上次增量备份以来的所有变化的数据。这种级别 1 的增量备份称为差异备份。还有一种级别 1 的增量备份，即累积备份，每次实现增量备份时，它总是备份自级别 0 备份以来所有变化的数据。显然差异增量备份会有多个备份文件，而累积增量备份只有一个备份文件，所以使用累积增量备份可以减少数据库的恢复时间。下面是使用级别 1 增量备份的实例。

【实例 26-24】使用 RMAN 实现增量备份的级别 1 备份。

```
    RMAN> backup incremental level 1 database;

    启动 backup 于 01-9月 -09
    使用通道 ORA_DISK_1
    通道 ORA_DISK_1: 启动增量级别 1 数据文件备份集
    通道 ORA_DISK_1: 正在指定备份集中的数据文件
    输入数据文件 fno=00002 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
    输入数据文件 fno=00001 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
    输入数据文件 fno=00003 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
    输入数据文件 fno=00005 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
    输入数据文件 fno=00004 name=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
    略过数据文件 00004，因为它未更改
    通道 ORA_DISK_1: 正在启动段 1 于 01-9月 -09
    通道 ORA_DISK_1: 已完成段 1 于 01-9月 -09
    段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\
    ORCL\BACKUPSET\ 2009_09_01\O1_MF_NNND1_TAG20090901T172008_59SST9QG_.B
    KP 标记=TAG20090901T172008 注释=NONE
    通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:55
    通道 ORA_DISK_1: 启动增量级别 1 数据文件备份集
    通道 ORA_DISK_1: 正在指定备份集中的数据文件
    备份集中包括当前控制文件
    在备份集中包含当前的 SPFILE
    通道 ORA_DISK_1: 正在启动段 1 于 01-9月 -09
    通道 ORA_DISK_1: 已完成段 1 于 01-9月 -09
    段句柄=F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\
    ORCL\BACKUPSET\ 2009_09_01\O1_MF_NCSN1_TAG20090901T172008_59SSW2PG_.B
    KP 标记=TAG20090901T172008 注释=NONE
    通道 ORA_DISK_1: 备份集已完成, 经过时间:00:00:03
    完成 backup 于 01-9月 -09
```

此时，完成了使用 RMAN 实现增量备份的所有步骤，在生产数据库中，读者可以使用操作系统工具编写脚本文件，从而实现增量备份的自动化。

26.2.5 使用 RMAN 实现脚本备份

对于长指令的 RMAN 备份操作，Oracle 提供了脚本语言功能，使得用户对特定的任务编写备份脚本，然后将脚本存储在恢复目录或存储为文本文件。下面通过实例说明如何创建和使用脚本

【实例 26-25】创建 RMAN 备份脚本。

```
D:\>rman catalog rman_backup/rman@orcl target system/oracle@orcl

恢复管理器: Release 11.1.0.6.0 - Production on 星期四 9 月 3 09:33:07 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到目标数据库: ORCL (DBID=1219822601)
连接到恢复目录数据库

RMAN> create script rman_backup{
2> sql 'alter system checkpoint';
3> backup database format
4> 'f:\offline_backup\back_%u.dbf';
5> backup current controlfile format
6> 'f:\offline_backup\back_ctl_%u.dbf';
7> }

已创建脚本 rman_backup
```

注意在创建 RMAN 备份脚本时必须连接到恢复目录和目的数据库，否则不能创建成功。

执行 RMAN 脚本，此时使用 RUN 指令和 EXECUTE SCRIPT 指令执行创建的脚本，其实执行脚本的过程就是执行一系列的 SQL 语句的过程，RMAN 脚本类似于 Windows 中的批处理文件，如下例所示。

【实例 26-26】执行脚本。

```
RMAN> run {execute script rman_backup;}

正在执行脚本: rman_backup
sql 语句: alter system checkpoint
启动 backup 于 03-9 月 -09
.........
```

在介绍完了如何创建脚本以及执行脚本后，下面介绍如何使用操作系统文件存储 RMAN 指令，并在 RMAN 中直接调用该文件执行 RMAN 命令。首先创建一个 rman_backup.rcv 文件，如图 26-2 所示。

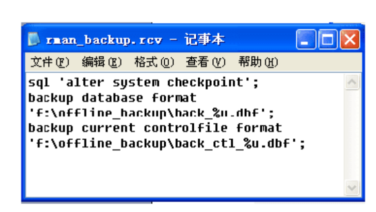


图 26-2　创建执行 RMAN 指令的操作系统文件

然后将该文件保存在 D 盘的根目录下，如实例 26-27 所示，在 RMAN 中调用操作系统文件执行 RMAN 指令。

【实例 26-27】调用操作系统文件执行 RMAN 指令。

```
D:\>rman catalog rman_backup/rman@orcl target system/oracle@orcl cmdfile
'rman_backup.rcv'

恢复管理器: Release 11.1.0.6.0 - Production on 星期四 9月 3 10:05:45 2009

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

连接到目标数据库: ORCL (DBID=1219822601)
连接到恢复目录数据库

RMAN> sql 'alter system checkpoint';
2> backup database format
3> 'f:\offline_backup\back_%u.dbf';
4> backup current controlfile format
5> 'f:\offline_backup\back_ctl_%u.dbf';
6>
7>
sql 语句: alter system checkpoint

启动 backup 于 03-9月 -09
分配的通道: ORA_DISK_1
…………
完成 Control File and SPFILE Autobackup 于 03-9月 -09
恢复管理器完成。
```

为了编辑方便，也可以将存储在恢复目录中的脚本文件转换为操作系统文件，如实例 26-28 所示。

【实例 26-28】将脚本文件转换为操作系统文件。

```
RMAN> print script rman_backup to file 'rman_backup.txt';

已将脚本 rman_backup 写入文件 rman_backup.txt
```

26.3 使用 RMAN 实现恢复

26.3.1 使用 RMAN 实现脱机备份的恢复

下面以恢复整个数据库为例说明在非归档模式下使用 RMAN 实现脱机备份的恢复。此时，自脱机备份以来变化的数据可能部分丢失，联机重做日志文件是循环使用的，一旦写满一个日志文件将切换到下一个，新的循环开始将覆盖掉部分变化的数据。这样的恢复其实是不完全恢复，因为数据库工作在非归档模式下。下面给出具体的步骤说明这种情况下如何恢复整个数据库。

01 *把数据库启动到 NOMOUNT 状态，以下实例假设数据库在运行，所以先关闭数据库，然后使用 NOMOUNT 参数启动数据库。*

【实例 26-29】将数据库启动到 NOMOUNT 状态。

```
RMAN> startup nomount;

已连接到目标数据库 (未启动)
Oracle 实例已启动

系统全局区域总计      603979776 字节

Fixed Size                    1260380 字节
Variable Size               218106804 字节
Database Buffers            377487360 字节
Redo Buffers                  7135232 字节
```

02 从备份的控制文件中恢复控制文件。

【实例 26-30】恢复控制文件。

```
RMAN> restore controlfile from autobackup;

启动 restore 于 01-9月 -09
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=157 devtype=DISK

恢复区域目标: F:\oracle\product\10.2.0/flash_recovery_area
用于搜索的数据库名 (或数据库的唯一名称): ORCL
通道 ORA_DISK_1: 在恢复区域中找到自动备份
通道 ORA_DISK_1: 已找到的自动备份:
F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\AUTOBACKUP\2009_09_01\O
1_MF_S_69646
5171_59TDD3LK_.BKP
通道 ORA_DISK_1: 从自动备份复原控制文件已完成
输出文件名=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL01.CTL
输出文件名=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL02.CTL
输出文件名=F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\CONTROL03.CTL
完成 restore 于 01-9月 -09
```

03 在上例中，要求使用自动备份的控制文件来恢复当前的控制文件，在使用 RMAN 实现脱机备份时，已经将 RMAN 的 CONFIGURE CONTROLFILE AUTOBACKUP 设置为 ON，所以在数据库备份时已经自动备份了控制文件，所以上例的执行可以成功。

04 将数据库切换到 MOUNT 状态，例如：

```
RMAN> alter database mount;

数据库已装载
释放的通道: ORA_DISK_1
```

05 此时，打开了控制文件，但是这个控制文件是从控制文件的自动备份中恢复的。

06 重建数据库，例如：

```
MAN> restore database ;
```

```
启动 restore 于 01-9月 -09
使用通道 ORA_DISK_1

通道 ORA_DISK_1: 正在开始恢复数据文件备份集
通道 ORA_DISK_1: 正在指定从备份集恢复的数据文件
正将数据文件 00001 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
正将数据文件 00003 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
…………
通道 ORA_DISK_1: 恢复完成, 用时: 00:01:26
完成 restore 于 01-9月 -09
```

07 打开数据库完成数据库恢复，例如:

```
RMAN> alter database open;

数据库已打开
```

26.3.2 使用 RMAN 实现脱机备份的恢复

在归档模式下，使用 RMAN 的脱机备份、所有归档重做日志以及当前的重做日志文件可以实现数据库的完全恢复。要求在使用 RMAN 脱机备份以来数据库一直运行在归档模式，且归档文件以及重做日志文件没有损坏。

下面以恢复整个数据库为例进行说明，其步骤如下所示。

01 如果数据库在运行，则首先使用 SHUTDOWN IMMEDIATE 指令关闭数据库。

【实例 26-31】在 RMAN 中关闭数据库并启动到 MOUNT 状态。

```
RMAN> startup mount;

已连接到目标数据库 (未启动)
Oracle 实例已启动
数据库已装载

系统全局区域总计      603979776 字节

Fixed Size                  1260380 字节
Variable Size             268438452 字节
Database Buffers          327155712 字节
Redo Buffers                7135232 字节
```

02 此时需要重建（RESTORE）数据库，从最近备份的全库备份集中重建整个数据库，其过程是将备份集中的数据文件拷贝到它原来的目录下，但是备份集的数据格式只有 RMAN 可以识别。

【实例 26-32】使用 RMAN 重建数据库。

```
RMAN> restore database;

启动 restore 于 01-9月 -09
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=156 devtype=DISK
```

```
通道 ORA_DISK_1: 正在开始恢复数据文件备份集
通道 ORA_DISK_1: 正在指定从备份集恢复的数据文件
正将数据文件 00001 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
正将数据文件 00002 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF
正将数据文件 00003 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
正将数据文件 00004 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF
正将数据文件 00005 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
………..
通道 ORA_DISK_1: 恢复完成, 用时: 00:02:15
完成 restore 于 01-9 月 -09
```

03 恢复（RECOVER）数据库，例如：

```
RMAN> recover database;

启动 recover 于 01-9 月 -09
使用通道 ORA_DISK_1
通道 ORA_DISK_1: 正在开始恢复增量数据文件备份集
通道 ORA_DISK_1: 正在指定从备份集恢复的数据文件
…………
正在开始介质的恢复
介质恢复完成, 用时: 00:00:06

完成 recover 于 01-9 月 -09
```

使用 RECOVER DATABASE 的目的是将在重建数据库时使用的备份集以来所有的用户提交数据重写入数据文件，完成数据库的完全恢复。此时，RMAN 会根据恢复过程应用相应的备份集或者归档日志文件。

04 打开数据库完成全库的完全恢复。

【实例 26-33】使用 RMAN 将数据库从 MOUNT 状态切换到 OPEN 状态。

```
RMAN> alter database open;

数据库已打开
```

26.3.3　使用 RMAN 从联机热备份中恢复

RMAN 支持从联机热备份中恢复整个数据库、一个表空间以及一个数据文件。若恢复整个数据库，则其方式和使用 RMAN 从脱机冷备份恢复（ARCHIVELOG 模式）步骤相同。如果是恢复表空间或者仅仅恢复一个数据文件，步骤基本相同，但是重建和恢复这些数据库对象略有区别，下面分别介绍这种情况下如何恢复一个表空间以及如何恢复一个数据文件。

1. 从 RMAN 联机备份中恢复表空间

（1）启动 RMAN 并建立与数据库服务器的连接

【实例 26-34】启动 RMAN。

```
D:\>rman target system/oracle@orcl
```

```
恢复管理器: Release 11.1.0.6.0 - Production on 星期二 9月 1 23:09:23 2009

Copyright (c) 1982, 2007, Oracle. All rights reserved.

连接到目标数据库: ORCL (DBID=1219822601)
```

（2）将需要恢复的表空间脱机

本例中将恢复 SYSAUS 表空间，例如：

```
RMAN> sql 'alter tablespace sysaux offline';

使用目标数据库控制文件替代恢复目录
sql 语句: alter tablespace sysaux offline
```

（3）重建表空间 SYSAUX

使用 RESTORE TABLESPACE 指令从最近备份的与该表空间相关的文件，这个过程（如找到最近备份集以及备份集中与需重建表空间相关的数据文件等）都由 RMAN 自动完成。

【实例 26-35】重建表空间 SYSAUX。

```
RMAN> restore tablespace sysaux;

启动 restore 于 01-9月 -09
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=144 devtype=DISK

通道 ORA_DISK_1: 正在开始恢复数据文件备份集
通道 ORA_DISK_1: 正在指定从备份集恢复的数据文件
正将数据文件 00003 恢复到 F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF
通道 ORA_DISK_1: 正在读取备份段 F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\
ORCL\BACKUPSET\2009_09_01\O1_MF_NNNDF_TAG20
090901T223404 59TD6X2G .BKP
通道 ORA_DISK_1: 已恢复备份段 1
段句柄 = F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\
BACKUPSET\2009 09 01\O1 MF NNNDF TAG20090901T223404 59TD6X2G
.BKP 标记 = TAG20090901T223404
通道 ORA_DISK_1: 恢复完成, 用时: 00:00:55
完成 restore 于 01-9月 -09
```

（4）恢复表空间

此时使用 RECOVER TABLESPACE 指令，因为备份集中表空间 SYSAUX 中的数据文件不是最新的，使用 RECOVER 将该表空间中变化的数据重新写入。

【实例 26-36】恢复表空间。

```
RMAN> recover tablespace sysaux;

启动 recover 于 01-9月 -09
使用通道 ORA_DISK_1
```

```
正在开始介质的恢复
介质恢复完成，用时: 00:00:03

完成 recover 于 01-9月 -09
```

（5）将表空间联机

例如：

```
RMAN> sql 'alter tablespace sysaux online';

sql 语句: alter tablespace sysaux online
```

至此，完成了对表空间 SYSAUX 的基于联机 RMAN 备份的恢复。

2. 使用 RMAN 恢复数据文件

恢复数据文件的过程和恢复表空间的过程一样，不给出具体的实例，只给出每一步的指令，读者可以当做作业来完成。

启动 RMAN

```
D:\rman target system/oracle@orcl
```

将要恢复的数据文件脱机：

```
RMAN>sql 'alter datafile f:\oracle\product\10.2.0\oradata\orcl\sysaux01.dbf
offline';
```

重建数据文件：

```
RMAN>restore datafile 'f:\oracle\product\10.2.0\oradata\orcl\sysaux01.dbf';
```

恢复数据文件：

```
RMAN>recover datafile 'f:\oracle\product\10.2.0\oradata\orcl\sysaux01.dbf';
```

将数据文件联机。

```
RMAN> sql 'alter datafile f:\oracle\product\10.2.0\oradata\orcl\sysaux01.dbf
online';
```

26.4　RMAN 的指令

26.4.1　VALIDATE BACKUPSET 指令

在使用 RMAN 备份了数据库后，需要恢复时最好使用 VALIDATE BACKUPSET 指令验证备份文件的可用性，如备份的数据文件都以备份集的形式存在，在使用 VALIDATE BACKUPSET 验证备份集时 RMAN 会自动找到指定的备份集，如实例 26-37 所示。

【实例 26-37】使用 VALIDATE BACKUPSET 指令验证备份集的可用性。

```
RMAN> validate backupset 30;
```

```
分配的通道: ORA_DISK_1
通道 ORA_DISK_1: sid=140 devtype=DISK
通道 ORA_DISK_1: 正在启动数据文件备份集验证
通道 ORA_DISK_1: 正在读取备份段 F:\ORACLE\PRODUCT\10.2.0\FLASH_
RECOVERY_AREA\ORCL\AUTOBACKUP\2009_09_01\O1_MF_S_69646757
3 59TGHPK2 .BKP
通道 ORA_DISK_1: 已恢复备份段 1
段句柄 = F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\AUTOBACKUP\
2009_09_01\O1_MF_S_696467573_59TGHPK2_.BKP 标记 =
TAG20090901T231263
通道 ORA_DISK_1: 验证完成, 用时: 00:00:02
```

在上例中，RMAN 确实启动了数据文件备份集验证，而且“验证完成”说明此备份集是有效地，可用于恢复操作。

读者应该会问数字 30 代表什么？其实它是 RMAN 的所有备份集中代表某个备份集的关键字，使用如下 LIST BACKUP SUMMARY 指令查看备份集的汇总信息。

【实例 26-38】查看备份集汇总信息。

```
RMAN> list backup summary;


备份列表
===============
关键字 TY LV S 设备类型    完成时间      段数 副本数  压缩标记
---------- --- --- -- --------------------- --------------------- -------- -
1       B  F  A DISK       22-8 月 -09 1     1  NO      TAG20090822T215265
…………
30      B  F  A DISK       01-9 月 -09 1     1  NO      TAG20090901T231263
31      B  F  A DISK       01-9 月 -09 1     1  NO     TAG20090901T232648
```

26.4.2 RESTORE…VALIDATE 指令

RMAN 支持使用 RESTORE…VALIDATE 验证数据库对象是否在当前的备份集中，这样在用户恢复一个数据文件或一个表空间时，可以首先确认该对象备份信息是否存在。

【实例 26-39】验证表空间 SYSAUX 备份信息是否在备份集中。

```
RMAN> restore tablespace sysaux validate;

启动 restore 于 02-9 月 -09
使用通道 ORA_DISK_1

通道 ORA_DISK_1: 正在启动数据文件备份集验证
通道 ORA_DISK_1: 正在读取备份段
…………
通道 ORA_DISK_1: 验证完成, 用时: 00:00:40
完成 restore 于 02-9 月 -09
```

下面再给出验证一个数据文件是否在备份集中的实例。

【实例 26-40】验证数据文件是否在备份集中。

```
    RMAN> restore datafile 'F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF'
validate;

    启动 restore 于 02-9月 -09
    使用通道 ORA_DISK_1

    通道 ORA_DISK_1: 正在启动数据文件备份集验证
    通道 ORA_DISK_1: 正在读取备份段
    …………
    通道 ORA_DISK_1: 验证完成, 用时: 00:00:47
    完成 restore 于 02-9月 -09
```

上述输出中的“验证完成”说明该数据文件存在于备份集中。

26.4.3　RESTORE…PREVIEW 指令

用户在备份数据库前，或许想知道执行恢复的所有文件是否存在，如当恢复表空间时，想知道该表空间中的所有数据文件是否在备份集中等。RMAN 提供了 RESTORE…PREVIEW 指令来完成这项功能，如实例 26-41 所示，查看恢复整个数据库所需的备份文件是否存在。

【实例 26-41】查看备份文件是否存在。

```
    RMAN> restore database preview;

    启动 restore 于 02-9月 -09
    使用通道 ORA_DISK_1
    备份集列表
    ===================
    …………
      备份集 27 中的数据文件列表
      文件 LV 类型 Ckp SCN    Ckp 时间    名称
      ---- -- ---- ---------- ---------- ----
      1       Full 4921182    01-9月 -09
    F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF
      …………….
      5       Full 4921182    01-9月 -09
    F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF
    已存档的日志副本列表
    关键字      Thrd Seq      S 短时间      名称
    ------- ---- ------- - ---------- ----
    ………….
    16       1    1       A 01-9月 -09
    F:\ORACLE\PRODUCT\10.2.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2009_09_02\O
    1 MF 1 1 59VF
    OKQG_.ARC
    介质恢复启动 SCN 是 4921182
    完成 restore 于 02-9月 -09
```

最后显示的“完成 restore”说明数据库所需要的备份文件都存在。通过以下方法同样可以验

证恢复某个表空间或数据文件所需的备份文件是否存在，如下所示：

```
RMAN> restore tablespace sysaux preview;
RMAN> restore datafile 5 preview;
```

26.5　本章小结

本章介绍了 RMAN 备份与恢复方面的知识，RMAN 实现了数据库备份与恢复的自动化处理，并且具有支持增量备份、指令简洁、维护方便的优点。在数据库恢复方面，读者在掌握使用传统的 EXP/IMP 和 EXPDP/IMPDP 数据泵技术外，需要理解用户管理的脱机和联机备份与恢复，这是数据库备份与恢复中最简单的部分。重点是理解 RMAN 备份技术，掌握使用 RMAN 实现联机备份、脱机备份以及相应的恢复方法。理解在归档模式和非归档模式下，数据库恢复的本质区别。

第 27 章

优化的概述

优化就是有目的地更改系统的一个或多个数据库组件，使得系统的运行符合一定的目标要求，如适当增加重做日志组以减少重做日志的频繁切换。而对于 Oracle 数据库系统而言，优化就是进行有目的地调整组件，从而改善系统性能，这些性能指标主要是指减少系统响应时间、减少用户等待和增加数据库系统的吞吐量。

27.1 优化的方法

通常说“好”的数据库系统是不需要优化的，所以我们先说明什么是“好”的系统，一个设计良好的数据库系统主要是指其数据库物理和逻辑设计合理，编写的 SQL 语句符合规范并且执行效率高，运行数据库的硬件环境满足数据库系统的需要，并且 DBA 对数据库的各种文件做了冗余分布等，对于这样的数据库系统只要使用适当的工具监视数据库的运行，及时发现影响数据库效率的瓶颈，一般是不需要优化的，所以对于需要优化的系统就是那些“不好”的数据库系统，往往这些数据库系统的设计就有问题，如不恰当地使用大对象表、效率低下的 SQL 语句（如在 WHERE 子句中使用子查询，而这些语句往往会消耗 CPU 资源），并且运行数据库的硬件资源不足（如 CPU 数量不够，或者有多个 CPU 但是内存不足）等。这样的数据库系统在运行期间通常是需要优化的。

优化是一个系统行为，需要细致分析导致系统瓶颈的问题，如果只是一味地增加内存、增加 CPU 不会解决一切问题，而往往是糟糕的数据库设计使得系统效率低下甚至无法使用，尤其是随着数据量的增加，由数据库设计引起的性能瓶颈就更加明显。

优化要考虑的问题是全面的，要求 DBA 考虑系统的各个方面，如 SGA 中的各种缓冲池、数据文件的分布、操作系统的性能、应用程序的 SQL 语句、索引的合理性等。总之需要分析引起性能低下的各种现象，通过适当的工具或数据字典来细致分析，最后找到具体的性能瓶颈，从而使用相应的优化方法来调整数据库组件或参数改善系统性能。

既然优化是一种系统行为，所以必然遵循一定的优化方法，一个好的优化方法应该是有条理、有顺序、基于优化目标的过程，虽然某个部分需要重复执行，经过反复调整以满足性能指标，但是面向目标是不会变的，即优化始终以适度的优化目标为基础。

优化方法决定了优化数据库的效率，一个好的优化方法其实是处理问题的一个逻辑思路，通

过它可以快速且准确地定位性能瓶颈。

一个良好方法应该是有适当的性能目标、对组件的更改可控制及实现跟踪、评估当前性能和性能目标之间的差距，直到满足最终的性能目标。

优化按照数据库系统使用前和使用后可以分为主动优化和被动优化，其中主动性能优化是指数据库系统应用前的性能优化或管理行为，如硬件选择、操作系统、性能和功能设计、存储系统选择、I/O 子系统配置和设计等，从而使得存储系统、数据库组件满足应用程序和 Oracle 的要求。被动性能优化，是指数据库系统应用后的性能优化或管理行为，涉及性能评估、故障排除、性能优化，在现有的硬件和软件体系结构内对系统环境进行优化。其实，在数据库维护中主要是如何使用被动的性能管理来优化系统。

下面给出一个数据库系统的优化方法，其实，它就是实施数据库性能优化的一个逻辑思路，将整个优化过程的框架记在心里，在优化时就时时知道每一步骤的目的，直到达到优化目标。

01 建立合理的优化目标。
02 确定当前系统性能。
03 确定性能瓶颈。
04 分析等待事件。
05 优化所需的数据库组件，如应用程序、SGA、I/O、LATCH 争用、OS 等。
06 跟踪调整后的性能是否满足优化目标。
07 重复步骤 3~步骤 6 直到满足优化。



在数据库优化时必须设置合理的优化目标，而且这个目标是量化的，不要过度优化，过度优化往往适得其反，不会有多少好的作用，一定要优化确认的性能瓶颈部分，不要优化不相干的组件，如果不作分析地增减 CPU 的数量，在某种条件下几乎无益于性能地提高，或许是内存太低造成的用户数据响应慢。

27.2 优化涉及的重要视图

在确定 Oracle 数据库的性能瓶颈时，数据字典视图提供关于当前系统状况的丰富信息，如数据字典视图 v$SYSTEM_EVENT、v$SESSION_EVENT、v$SESSION_WAIT 就是确认性能瓶颈的几个重要视图。

27.2.1 v$SYSTEM_EVENT 视图

从该视图的名字可以看出，这是个系统事件视图，其作用是记录自实例启动以来的等待事件，它记录了事件名称、事件的总等待次数、事件的等待超时次数、对某个事件的总等待时间以及平均等待时间等信息，通过这些信息，找到等待时间较多的事件，查看是基于 I/O 事件还是基于内存的事件，这里的数据是“历史数据”，也就是说它是事件的累计记录，不代表当前的会话事件。通过 DESC 查看其结构信息，如实例 27-1 所示。

【实例 27-1】数据字典 v$SYSTEM_EVENT 的结构。

```
SQL> connect system/oracle@orcl
已连接。
SQL> desc v$system_event;
 名称                                      是否为空? 类型
 ----------------------------------------------------------------- -
 EVENT                                     VARCHAR2(64)
 TOTAL_WAITS                               NUMBER
 TOTAL_TIMEOUTS                            NUMBER
 TIME_WAITED                               NUMBER
 AVERAGE_WAIT                              NUMBER
 TIME_WAITED_MICRO                         NUMBER
 EVENT_ID                                  NUMBER
 WAIT_CLASS_ID                             NUMBER
 WAIT_CLASS#                               NUMBER
 WAIT_CLASS                                VARCHAR2(64)
```

下面重点解释几个重要的列属性，并对重要的列属性给出示例解释。

1. EVENT

用于记录事件名称，如数据库文件离散读、数据库文件顺序读、缓冲区忙等待、闩锁空闲、空闲缓冲区等待等。

通过实例 27-2 查看当前系统上有多少不同的等待事件，随后试着解释其中的部分重要等待事件。

【实例 27-2】查询系统上的不同的等待事件名。

```
SQL> col event for a50
SQL> select distinct(event)
  2* from v$system_event

EVENT
--------------------------------------
os thread startup
control file parallel write
checkpoint completed
log file sync
db file parallel write
SQL*Net message from client
LGWR wait for redo copy
control file sequential read
read by other session
virtual circuit status
dispatcher timer

EVENT
--------------------------------------
SQL*Net break/reset to client
pmon timer
rdbms ipc message
```

```
buffer busy waits
log file sequential read
db file sequential read
PX Deq: Par Recov Reply
smon timer
......
已选择 57 行。
```

等待事件分为两种，即空闲等待和非空闲等待，其中空闲事件，如 smon timer（smon 计时器）、pmon timer（pmon 计时器）、rdbms ipc message（rdbms 的 ipc 消息）、SQL*Net message from client（来自客户机的 SQL*Net 消息）。而等待事件，如 buffer busy waits（缓冲区忙等待）、log file sequential read（日志文件顺序读等待）、db file sequential read（数据文件顺序读等待）、db file parallel write（数据文件并行写等待）、control file parallel write（控制文件并行写等待）、log file sync（日志文件同步）等。

在熟悉了等待事件后，就可以很容易地判断哪个等待事件造成了系统性能的下降，从而确定性能瓶颈所在。

2. TOTAL_WAITS

所有会话对于某一事件的总的等待次数。

3. TOTAL_TIMEOUTS

某一事件等待超时的次数。

4. TIME_WAITED

所有会话对于某一事件的总等待时间，单位是 0.01s。

5. AVERAGE_WAIT

所有会话对某一事件的平均等待时间，其中 AVERAGE_WAIT= TIME_WAITED/TOTAL_WAITS，单位是 0.01s。

6. WAIT_CLASS#

等待类型号，唯一标识一种等待类型，如下例所示。

【实例 27-3】查询等待类型号。

```
SQL> select distinct(wait_class#)
  2  from v$system_event;

WAIT_CLASS#
-----------
         1
         6
         2
         4
         5
         8
```

```
          7
          9
          0

已选择 9 行。
```

可见当前查询到 9 个等待类型号。

7. WAIT_CLASS

等待类型，如 IDLE 空闲等待、SYSTEM I/O 系统 I/O、USER I/O 用户 I/O 以及 CONCURRENCY 并发等待等。下面查询 WAIT_CLASS 的类型。

【实例 27-4】查询系统等待类型。

```
SQL> select distinct(wait_class)
  2  from v$system_event;

WAIT_CLASS
------------------------------------
Concurrency
System I/O
User I/O
Configuration
Other
Application
Idle
Commit
Network

已选择 9 行。
```

那么可以通过 WAIT_CLASS#和 WAIT_CLASS 来查找特定的等待类型号对应的等待事件类型，如下例所示。

【实例 27-5】查询特定等待类型号对应的等待类型。

```
SQL> select distinct(wait_class)
  2  from v$system_event
  3  where wait_class# = 9;

WAIT_CLASS
-----------------------------------------
System I/O
```

可见，WAIT_CLASS#=9 的等待事件名为 System I/O，即系统 I/O。同理可以查询其他等待类型号对应的等待类型。

下面列出经常遇到的一些等待事件，如表 27-1 所示。读者熟悉这些等待事件是优化工作的基本前提，读者必须理解这些事件的内在含义。

表 27-1　等待事件列表

等待事件名称	等待事件含义
db file scattered read （数据库文件离散读）	该等待事件也是普遍存在的一个等待事件，该事件说明用户进程正从数据文件读数据到数据库缓存中，等待 I/O 数据的返回，scattered read 就是 full scan 的意思，因为执行 full scan 时连续的数据读到内存时，并不是顺序存放的。存在由于全表扫描而引起的等待，I/O 争用或过多 I/O
Db file sequential read （数据库文件顺序读）	这是一个相对普遍的 I/O 等待时间，通常与单个数据块的读取操作相关联，说明存在索引扫描相关的等待，索引扫描一般会减少数据访问事件，所以该事件的出现说明存在 I/O 争用或存在过多 I/O 操作，如多个用户同时访问一个数据文件
Direct path read （直接路径读）	直接路径读取发生在直接读数据到 PGA 时，因为数据不经过 SGA，所以称为直接路径读。这种等待事件意味着设备上的 I/O 竞争，导致在直接路径读取时的等待
Direct path write （直接路径写）	直接路径写发生在从 PGA 直接写数据到数据文件时，因为数据不经过 SGA，所以称为直接路径写。该等待说明存在 I/O 争用，应该找到 I/O 操作频繁的数据文件，分散负载
Log file switch （archiving switch，即 日志文件切换事件，归档切换）	对于处于归档模式的数据库而言，当日志组写满后，在日志切换时，如果需要覆盖先前的日志，而该日志需要归档进程写入归档文件，由于写入归档文件需要时间，在此期间就产生了 Log file switch 事件，该等待事件的原因一般是由于 I/O 问题、ARCH 归档进程跟不上 LGWR 日志写进程的速度或者日志组太少引起的
Log file switch （checkpoint incomplete，即 日志文件切换事件，检验点未完成）	在重做日志组写满后，需要覆盖先前的日志文件，当该日志文件保护的脏数据还没有完全写入数据库时，出现该等待事件，等待检验点的完成，该等待事件说明 DBWR 进程写入速度慢或存在数据文件的 I/O 问题，可以采用增加 DBWR、增加日志组和修改日志文件大小的方式解决
Log file sync （日志文件同步）	在用户提交或回滚数据时，LGWR 进程需要将重做日志缓冲区中的数据写入重做日志，当 LGWR 完成该任务后就通知相应的用户进程，如果这个写入过程很长就导致了 log file sync 事件，因为日志文件同步过程需要对 LGWR 的写入任务完成。该等待事件说明 LGWR 的写入速度慢，或有大量的用户提交或回滚数据。解决方式主要是提高 LGWR 的性能，将重做日志文件放在高速盘上，用户尽量使用批量作业等
Log file single write	该等待事件仅与写日志文件头块有关，表示检查点中的等待

（续表）

等待事件名称	等待事件含义
Log file parallel write	从重做日志缓冲区将重做记录写入磁盘时，如果重做日志组有多个日志成员，则会把重做记录同步写入这些日志成员，该操作是并行的，如果存在 I/O 问题则会发生该等待
Log buffer space（日志缓冲区空间等待事件）	该事件的含义是日志缓冲区空间不足，说明数据库产生重做日志的速度比 LGWR 的速度快，对于产生重做日志的进程而言就认为是日志缓冲区不足，在日志切换太慢时也会出现该等待事件，因为切换期间产生的重做日志无法写入日志文件，只能等待。解决方法是增大重做日志缓冲区的尺寸或者增大重做日志文件的大小。也可以提高 LGWR 的速度，折中的方式是把日志文件放在高速磁盘上相当于提高了 LGWR 的速度
Latch free	该等待事件是 Oracle 的某个内存结构无法获取时产生的，Oracle 使用 latch 来保护共享内存的串行访问。通常可以将 latch 的数量调整到最大数目。如果问题依然没有消除就需要确定是什么类型的闩锁，做进一步判断该内存结构为什么存在争用

27.2.2　v$SESSION_EVENT 视图

该视图提供了当前活动会话的事件信息，当系统性能发生下降时，查看该视图可以发现一些重要的事件和该事件相关的信息。

【实例 27-6】查看数据字典视图 v$SESSION_EVENT 的结构。

```
SQL> desc v$session_event;
 名称                                       是否为空? 类型
 -------------------------------------------------------------
 SID                                        NUMBER
 EVENT                                      VARCHAR2(64)
 TOTAL_WAITS                                NUMBER
 TOTAL_TIMEOUTS                             NUMBER
 TIME_WAITED                                NUMBER
 AVERAGE_WAIT                               NUMBER
 MAX_WAIT                                   NUMBER
 TIME_WAITED_MICRO                          NUMBER
 EVENT_ID                                   NUMBER
 WAIT_CLASS_ID                              NUMBER
 WAIT_CLASS#                                NUMBER
 WAIT_CLASS                                 VARCHAR2(64)
```

比较实例 27-1 和实例 27-5 可以看出数据字典 v$session_event 比数据字典 v$system_event 多了两个列属性：sid 会话 ID 和 max_wait。而其他参数的含义二者相同。如果在系统级和会话级都发现某一个事件有问题，往往有助于确定造成系统性能的事件。下面通过一个实例查询当前数据系统的会话等待事件信息。

【实例 27-7】通过数据字典 v$session_event 查看当前的会话事件信息。

```
SQL> col event for a30
SQL> select sid,event,total_waits,wait_class
  2  from v$session_event
  3  where event like 'db%';

      SID EVENT                          TOTAL_WAITS WAIT_CLASS
-------------- ------------------------------------------------ -
      147 db file sequential read                  2          User I/O
      154 db file sequential read                 27          User I/O
      160 db file sequential read                  1          User I/O
      161 db file sequential read                462          User I/O
      161 db file scattered read                   6          User I/O
      162 db file sequential read                 31          User I/O
      163 db file sequential read                  2          User I/O
      164 db file sequential read               3052          User I/O
      164 db file scattered read                  93          User I/O
      165 db file sequential read                  6          User I/O
      167 db file parallel write                 164          System I/O
```

视图 v$session_event 虽然是活动的会话信息，但它是累计结果，自实例启动以来的某一事件发生的多次等待会被累计。如果需要知道某个事件段内事件的增长幅度（大变化幅度的等待事件往往显示一些性能问题）可以使用如下方法，即先创建当前视图的一个表，等待一段时间再创建一个当前视图的表，通过这两个表的操作来完成某一个时间段内事件的增长幅度。

【实例 27-8】创建一个起始时间临时表。

```
SQL> create table begin_sesseion_event
  2  as
  3  select *
  4  from v$session_event;

表已创建。
```

该表创建完后，等待一段时间（根据业务需要选择），如 10 分钟，然后创建第二临时表。

【实例 27-9】创建一个终止时间临时表。

```
SQL> create table end_sesseion_event
  2  as
  3  select *
  4  from v$session_event;

表已创建。
```

然后通过两个表的连接来查看 10 分钟内当前会话中事件的属性值增幅信息。

【实例 27-10】查看 10 分钟内会话中事件的属性值增幅信息。

```
SQL> select begin.sid,begin.event,(begin.total waits-end.total waits) waitsin10m,
begin.wait_class
```

```
  2  from begin_sesseion_event begin,end_sesseion_event end
  3  where begin.sid = end.sid
  4  and begin.event like 'db%';

       SID EVENT                          WAITSIN10M WAIT_CLASS
---------- ------------------------------ ---------- --------------------
       147 db file sequential read                 1 User I/O
       147 db file sequential read                 0 User I/O
       154 db file sequential read                42 User I/O
       154 db file sequential read                51 User I/O
       ..........
       167 db file parallel write                410 System I/O

已选择 53 行。
```

上例中，我们查询了事件名内包含 db 的等待事件在 10 分钟内的 total_waits 的增幅信息。注意通过这里的会话 ID 和 EVENT 列结合下面讲的 V$SESSION_WAIT 视图可以确定等待事件，从而锁定性能瓶颈。

27.2.3 v$SESSION_WAIT 视图

v$SESSION_WAIT 视图是实时的视图，每次查询的结果或许都不同，基于会话级的等待事件，该视图记录了等待事件和相应资源的更详细的信息。

【实例 27-11】查看视图 v$SESSION_WAIT 的结构。

```
SQL> desc v$session_wait;
 名称                                        是否为空? 类型
 ----------------------------------------------------------------- --
 SID                                         NUMBER
 SEQ#                                        NUMBER
 EVENT                                       VARCHAR2(64)
 P1TEXT                                      VARCHAR2(64)
 P1                                          NUMBER
 P1RAW                                       RAW(4)
 P2TEXT                                      VARCHAR2(64)
 P2                                          NUMBER
 P2RAW                                       RAW(4)
 P3TEXT                                      VARCHAR2(64)
 P3                                          NUMBER
 P3RAW                                       RAW(4)
 WAIT_CLASS_ID                               NUMBER
 WAIT_CLASS#                                 NUMBER
 WAIT_CLASS                                  VARCHAR2(64)
 WAIT_TIME                                   NUMBER
 SECONDS_IN_WAIT                             NUMBER
 STATE                                       VARCHAR2(19)
```

这里需要介绍几个列属性及其含义。

- EVENT：等待事件的名称，常见的等待事件，如数据文件离散读、数据文件顺序读、空闲缓冲区等待、闩锁空闲或缓冲区忙等待等，因为视图 v$session_wait 反映了系统当前的等待事件，所以需要查找那些不断出现的等待事件，这些事件是分析的重点。
- P1~P3：参数 P1~P3 是我们要找的等待事件的详细信息，这些列的值是其他 V$视图的外部键，对每一个值的解释都与等待事件相关。

如对于数据文件离散读等待事件，其中 P1 表示文件号，P2 表示进程正等待的数据块号，P3 表示从 P2 指定的块号中读取的块数。此时通过参数 P1 的文件号，以及静态数据字典 DBA_DATA_FILES 可以确定哪个数据文件发生了数据库文件离散读等待。通过参数 P2 和静态数据字典 DBA_EXTENTS 可以确定等待数据块号对应的数据区段，以及该区段所属的用户和表空间，从而进一步确认发生等待事件的对象。

对于闩锁空闲事件，参数 P2 为闩锁号，它与数据字典 v$latch 中的 latch#相对应，通过查询 v$latch，就知道哪些闩锁有问题，闩锁保护的是内存区，所以需要调整内存的大小或者调整相关文件的 I/O 分布。图 27-1 显示了查询数据库系统上等待时间不为 0 的闩锁事件信息。

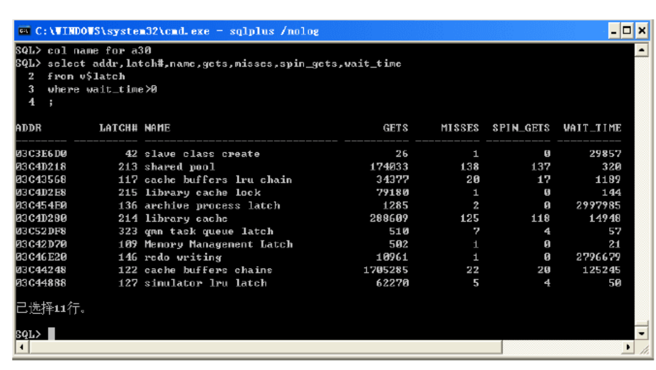


图 27-1 查询数据库系统上的闩锁事件信息

通过 NAME 列的等待闩锁的事件，查询相关的事件解释，找到系统性能瓶颈。

1. State

该参数说明给定事件的状态，有如下 4 种状态。

- Waiting：说明会话当前正在等待此事件。
- Waited short time：该参数值说明了会话已经等待了一段时间，但是这个等待时间的事件可以忽略，除非该事件重复出现。
- waited known time：说明某个进程请求它正等待的资源。
- waited unknown time：如果 TIMED_STATISTICS 设置为 FALSE，则 state 的值为该状态。

2. Wait_time

该值依赖于 state 参数的值，以秒为计算单位，如果 STATE 为 WAITING、WAITED UNKNOWN TIME、WAITED SHORT TIME，则 WAIT_TIME 的值为 irrelevant，如果 STATE 为 WAITED KNOWN TIME，则 WAIT_TIME 的值为具体的等待事件，但是前提是系统必须设置参数 TIMED_STATISTICS 为 TRUE。

3. Seconds_in_wait

该值依赖于 state 参数的值，以秒为计算单位。如果 STATE 为 WAITED UNKNOWN TIME、WAITED KNOWN TIME、WAITED SHORT TIME，则 WAIT_TIME 的值为 irrelevant，如果 STATE 为 WAITING，则 WAIT_TIME 的值为具体的等待事件。

在了解了几个重要视图以及视图的作用后，通过它们可以确定影响性能的瓶颈，在确认后就着手优化了，这里需要着重说明不要被较高的高速缓存命中率所迷惑，它往往掩盖了性能瓶颈的真相，要基于等待事件作为确定系统需要优化调整的依据，所以使用 Oracle 内的会话以及等待信息确定系统当前的性能，优化步骤如下所示。

01 从 v$system_event 确认实例自启动以来的所有等待事件，从中挑出那些具有较长等待时间的事件，如数据文件顺序读、空闲缓冲区或缓冲区忙等待等。

02 从 v$session_event 确定给定的事件，如缓冲区忙等待涉及哪些会话，注意该视图是当前所有会话的活跃等待信息，而 v$system_event 是历史数据的累计，后者具有参考价值，通过 v$system_event 中的典型事件再去查看事件设计的会话。

03 进一步查看 v$session_wait 视图，查看造成等待事件的具体信息，如哪些资源争用，是数据文件、缓冲区还是闩锁等。

04 查看 v$session_wait 视图中的 p1~p3 的值，分析与其他视图的关系，选取前 5 个事件重点分析直到找到瓶颈所在。

05 对具体瓶颈，如糟糕的 SQL 语句等确定优化方案。

06 观察性能指标，继续调整相关数据库组件直到满足系统性能需要。

27.3　优化的流程及相关说明

优化是一个迭代的过程，其基本思路是发现性能问题→通过各种视图或其他工具确定瓶颈→针对问题调整数据库组件→监控性能是否满足目标要求→重复调整。用流程图来说明这个调整过程，如图 27-2 所示。

以上的优化思路是一个逻辑框架，具有指导作用，但是不具有可操作性，优化是一个系统而具体的工作，所以下面给出优化更具体的单向流程图，如图 27-3 所示。

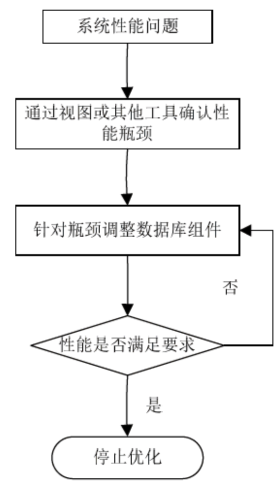


图 27-2　优化逻辑流程图

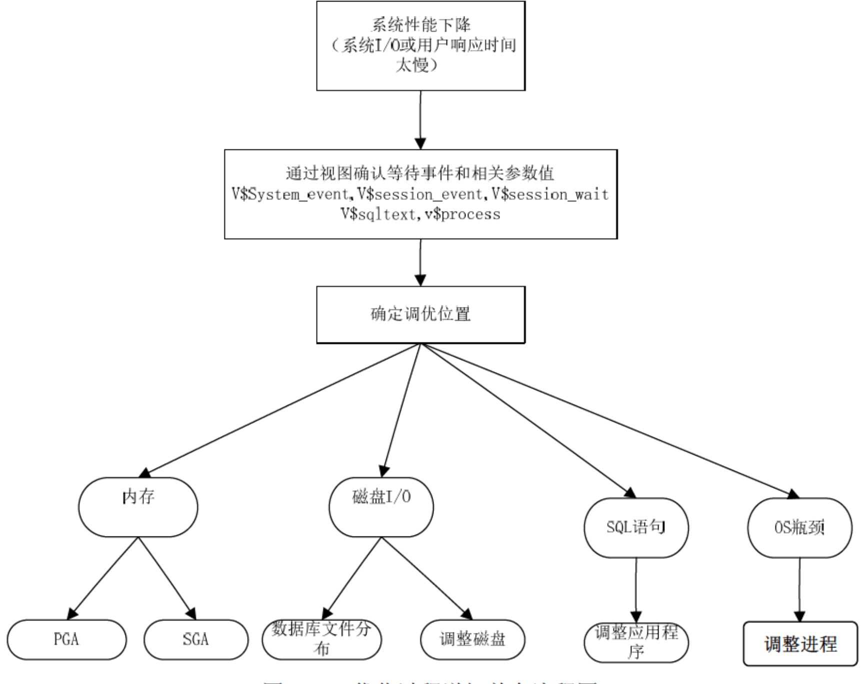


图 27-3　优化过程详细单向流程图

依据这个流程图来优化数据库，在讲解具体的组件优化时，按照这些组件分类介绍，其思路是通过系统的等待事件确认瓶颈所在，其实一个等待事件往往涉及多个组件的调整，如调整 DBWR 的个数，或许也需要调整文件到高速磁盘等，在介绍时，主要针对一个具体的组件做优化，但是希望读者理解这样的优化不是封闭的，优化是个开放的系统行为，需要多角度去考虑，针对等待事件调整相关的数据库组件。

同时，在介绍具体组件的优化时，可能没有给出具体的等待事件，但是只要读者知道这些组件的优化原则和优化方法，掌握这些优化的原子操作，对于一个具体的优化行为可以组合使用这些数据库组件的原子优化方法，往往可以很好地解决优化问题。

27.4　本章小结

Oracle 数据库性能优化是一个系统行为，我们始终要从全局的高度看待和分析性能瓶颈问题，基本的思路是：通过数据字典视图或监视工具来发现等待事件，分析各种类型等待事件导致的原因，适当调整参数或者尝试优化 SQL，观察性能问题是否继续存在，如果继续存在，则需要从其他角度考虑，或者继续通过等待事件分析原因直到性能满足用户需求。

第 28 章

SQL语句以及内存优化

DBA 是无法修改应用系统程序的，所以对于由于 SQL 语句造成的性能问题必须通过 DBA 可操作的资源来最大化地调整 SQL 语句的效率，而调整 SGA 则是 DBA 容易操作的内容，但是在 Oracle 11g 中，可以由数据库服务器自动调整，只有少数的参数还需要人工调整，学会适当调整这些参数，而不是高枕无忧地让数据库服务器去完成仍然是件好事。本章将分析 SQL 语句的方法、示例以及如何通过索引优化 SQL 语句，通过工具包将程序和数据常驻内存以减少物理读磁盘的时间，从而减少数据或程序的响应时间，最后介绍了优化重做日志缓冲区以及优化 PGA。

28.1 优化 SQL 语句

一个设计良好的应用程序是不需要优化的，这里的“良好”在很大程度上是指良好的 SQL 语句设计，在 Oracle 数据库应用系统中几乎有 80%的性能问题是由糟糕的 SQL 语句引起的。

那么如何写出“良好”的 SQL 语句呢？以下将给出一个条款式的说明，并作简单地讨论。

- 对于规模较小的表，如果 SQL 语句的 WHERE 子句有 GROUP BY、DISTINCT 或 GROUP BY，则对设计的列建立明确的索引。
- 如果 SQL 语句的 WHERE 子句在使用索引时比执行全表扫描还要耗时，则不使用索引。
- 在应用程序的 SQL 语句中不要使用相关子查询，因为随着子查询和主查询中表内行记录的增长，这种查询将极大地消耗 CPU 资源，而尽量使用联机视图重新编写。
- 在 SQL 语句的 WHERE 子句中用 not exists 来代替 not in。
- 使用 like 运算符代替 substr 函数，因为如“scot%”这样的 like 运算符将使用索引，而 substr 函数将使索引无效。
- 如果 SQL 语句中频繁使用基于某种计算规则的查询，并且知道所涉及的列，则创建基于函数的索引。
- 如果查询总是基于主从表关系的行，则对外键建立索引。
- 尽量减少建立索引的时间，可以将 sort_area_size 设置的足够大，使得建立索引的排序行为都在内存中发生，或者在系统不繁忙时创建大表的索引。

上述条款式说明都基于减少 I/O 和更有效的内存使用，通过减少系统 I/O、有效利用内存和 CPU 的使用减少系统压力，增加系统对用户的响应时间。

在理解了什么是好的 SQL 语句时，在遇到 SQL 优化的情况下就可依照这些原则进行修改，当

然如果应用程序已经使用，对于一个公司或企业为了优化而修改数据库设计中的问题是不太现实的，所以只好采用迂回的方式尽量避免糟糕的 SQL 语句对系统性能的影响。

如果通过数据字典视图或其他检测工具，发现造成等待事件的原因是一个 SQL 语句，如该语句对具有 1000000 行的数据不断地进行全表扫描，而且具有子查询。显然就需要优化这样的 SQL 语句。

28.1.1 获得 SQL 语句的执行计划

SQL 语句执行计划就是解释 SQL 语句的执行步骤，在 Oracle 中使用 explain plan for 指令来获得 SQL 语句的执行计划，也可以使用 autotrace 执行获得 SQL 语句的执行过程和相关的统计信息，如物理读的数据量、磁盘内排序的数据量等。下面分别介绍这两种常用的获得 SQL 语句执行计划的指令。

1. 使用 EXPLAIN PLAN FOR 命令

在使用该指令时，必须先使用一个脚本文件 utlxplan.sql 来创建 plan_table，从而存储使用 explain plan for 语句获得的 SQL 语句的分析结果。

其实 utlxplan.sql 就是创建表 PLAN_TABLE 的脚本程序，打开该文件可以清楚地看到创建表的说明和建表的 SQL 语句，如下所示。

```
Rem This is the format for the table that is used by the EXPLAIN PLAN
Rem statement.  The explain statement requires the presence of this
Rem table in order to store the descriptions of the row sources.

create table PLAN TABLE (
       statement_id      varchar2(30),
       plan_id          number,
       timestamp         date,
       remarks          varchar2(4000),
       operation         varchar2(30),
       options          varchar2(285),
       ………..省略了部分列属性的定义
       other_xml         clob
);
```

该脚本文件的说明部分也清楚地介绍了创建该表的目的就是存储对于执行计划的过程描述信息。在 Oracle 数据库中有很多类似的 SQL 脚本文件，只要打开这些文件，分析一下，一切都明晰了，并且读者也可以模仿写出规范的 SQL 脚本文件。

下面执行该脚本文件，如实例 28-1 所示。

【实例 28-1】执行脚本文件 utlxplan.sql。

```
SQL> @F:\oracle\product\10.2.0\db 1\RDBMS\ADMIN\utlxplan.sql

表已创建。
```

显然，表已经创建，在打开的 utlxplan.sql 脚本文件中可以看到表名为 PLAN_TABLE。查询该

表的结构，同时也是为了验证该表是否存在，如实例 28-2 所示。

【实例 28-2】查看表 PLAN_TABLE 的结构。

```
SQL> desc plan_table;
 名称                                        是否为空? 类型
 ------------------------------------------------------------- ------
 STATEMENT_ID                                          VARCHAR2(30)
 PLAN ID                                               NUMBER
 TIMESTAMP                                             DATE
 REMARKS                                               VARCHAR2(4000)
 OPERATION                                             VARCHAR2(30)
 OPTIONS                                               VARCHAR2(285)
 OBJECT_NODE                                           VARCHAR2(128)
 OBJECT_OWNER                                          VARCHAR2(30)
………..
TIME                                                   NUMBER(38)
 QBLOCK_NAME                                           VARCHAR2(30)
 OTHER_XML                                             CLOB
```

现在创建了为了执行 explain plan for 指令所需的表，就可以使用该指令执行 SQL 语句的分析了。通过实例 28-3 说明如何使用 explain plan for 指令分析 SQL 语句执行计划。

【实例 28-3】通过 explain plan for 指令分析 SQL 语句的执行计划。

```
SQL> explain plan for
  2  select count(*) from scott.emp;

已解释。
```

此时解释了 SQL 语句 select count(*) from scott.emp，该语句的作用是计算用户 SCOTT 的表中记录的行数。其执行分析信息存储在表 PLAN_TABLE 中，查询该语句的执行计划，如实例 28-4 所示。

【实例 28-4】查看表 PLAN_TABLE 中的 SQL 语句执行计划信息。

```
SQL> col id for 999
SQL> col operation for a20
SQL> col options for a20
SQL> col object_name for a20
SQL> select id,operation,options,object_name,position
  2  from plan table;

  ID OPERATION            OPTIONS         OBJECT_NAME   POSITION
------ -------------------------------- ------------------------ ----
   0 SELECT               STATEMENT                         3
   1 SORT                AGGREGATE                          1
   2 TABLE ACCESS         FULL              EMP              1
```

从上例的输出中可以看出 SQL 语句的执行过程，现在分析最后一行，ID 说明步骤标识，OPERATION 为 TABLEACCESS，说明该步骤的行为是访问表，OPTIONS 为 FULL，说明使用全

表扫描访问表，OBJECT_NAME 说明行为的对象为表 EMP。

在分析一个 SQL 语句时，经常使用该指令判断是否使用适当的索引完成表的访问，从而适当地修改索引或创建索引来优化 SQL 语句的执行。下面继续分析如何使用 AUTOTRACE 指令分析 SQL 语句的执行计划。

2. 使用 AUTOTRACE 命令

使用 AUTOTRACE 指令可以跟踪 SQL 语句并分析其执行步骤、统计信息，如物理读数据量、磁盘和内存排序数据量。但是要执行该指令需要设置几个参数。

- SQL_TRACE：该参数说明是否启动对 SQL 语句的追踪。默认该参数为 FALSE，要启用 AUTOTRACE 功能需要将参数 SQL_TRACE 设置为 TRUE，该参数可以动态改变。注意，在不需要追踪 SQL 语句时，最好将该参数设置为 FALSE，因为它会造成跟踪所有执行的 SQL 语句，这样会产生大量的 TRC 文件，对磁盘空间有一定的冲击。
- USER_DUMP_DEST：该参数说明 SQL 语句追踪文件的记录位置，在笔者的计算机上其默认目录为 F:\oracle\product\10.2.0\admin\orcl\udump。
- TIMED_STATISTICS：该参数可以使用 ALTER SYSTEM 或 ALTER SESSION 动态设置。默认参数值为 TRUE。

所以，我们只需要设置参数 SQL_TRACE 来启动对 SQL 语句执行的追踪，如实例 28-5 所示。

【实例 28-5】设置参数 SQL_TRACE 启动 SQL 语句追踪。

```
SQL> alter system set sql_trace = true;

系统已更改。
```

下面查询当前系统上是否启用 SQL 语句追踪功能，如实例 28-6 所示。

【实例 28-6】查询 SQL_TRACE 参数值。

```
SQL> show parameter sql_trace;

NAME                                 TYPE        VALUE
--------------------------------------------------------
sql_trace                            boolean     TRUE
```

现在，已经做好了使用 AUTOTRACE 指令查看 SQL 语句执行计划的准备，下面通过实例 28-7 说明如何使用该指令，并解释相关参数。

【实例 28-7】使用 AUTOTRACE 追踪 SQL 语句执行计划。

```
SQL> set autotrace traceonly;
SQL> select count(*) from scott.emp;


执行计划
----------------------------------------------------------
Plan hash value: 2083865914
```

```
-----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost  (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| EMP  |    56 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------


统计信息
-------------------------------------------------------------
       224  recursive calls
         0  db block gets
        28  consistent gets
         5  physical reads
         0  redo size
       408  bytes sent via SQL*Net to client
       385  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         4  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

上例中首先启动 AUTOTRACE 并仅执行 TRACE 功能、解释 EXPLAIN 和统计 STATISTICS 功能。AUTOTRACE 结果分为两部分，一部分是 SQL 语句的执行计划，另一部分是统计信息。

从执行计划可以看出 SQL 语句的执行步骤、访问的对象以及消耗的 CPU，如果是表，还记录访问的表的行数。

统计信息显示了更具体的数据访问和磁盘访问的细节，如物理读数据量、重做数据量、迭代访问传送到客户端的数据量，以及客户端传递给数据库服务器的数据量、内存排序的数据量，以及磁盘排序的数据量，其实如果此处出现磁盘排序，即 sorts(disk)的值不为 0，或许需要调整 PGA 中 SORT_AREA_SIZE 的尺寸（排序区自动管理的情况下），或调整 PGA_AGGREGATE_TARGET 参数以增减 PGA 的尺寸，使得排序行为尽可能在内存中完成。

下面详细介绍统计信息中每一行的含义。

- recursive calls：递归调用的次数。
- db block gets ：读数据块的数量。
- consistent gets：总的逻辑 I/O。
- physical reads：物理 I/O。
- redo size：重做数量。
- bytes sent via SQL*Net to client：SQL*Net 通信。
- sorts (memory)：内存排序统计。
- sorts (disk)：磁盘排序统计。
- rows processed：被检索的行数。

在不需要使用 AUTOTRACE 时，将该功能关闭，不然所有接下来执行的 SQL 语句都会被追踪分析，如实例 28-8 所示。

【实例 28-8】关闭 AUTOTRACE。

```
SQL> set autotrace off;
```

查看启动 AUTOTRACE 的一些参数，如实例 28-9 所示。

【实例 28-9】查看启动 AUTOTRACE 的其他参数。

```
SQL> set autotrace
用法: SET AUTOT[RACE] {OFF | ON | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]
```

如果需要启动 AUTOTRACE 功能但只需要执行计划，可使用 SET AUTOTRACE ON explain 指令；如果启动 AUTOTRACE 功能而只需要统计信息，可使用 SET AUTOTRACE ON statistics 指令；如果都需要，则可使用 SET AUTOTRACE TRACEONLY 指令。使用 SET AUTOTRACE ON 指令默认与使用 SET AUTOTRACE TRACEONLY 指令相同，都是启动追踪执行计划和追踪统计信息。

28.1.2 通过建立索引优化 SQL 语句

对于用户经常使用的查询，尤其是大表的查询，在程序设计时应该对这些问题做出预测，并要求对这样的查询尽量使用索引，这样通常可以加快查询速度、减少系统 I/O，但是并不是所有的系统都经过“精细”的需求分析，如果遇到这样的情况，发现某个 SQL 语句总是使用全表扫描实现用户的查询，则需要通过建立索引加快查询速度，这是 DBA 可以介入的优化 SQL 语句的方法，截断表的操作如实例 28-10 所示。

【实例 28-10】截断 PLAN_TABLE 表。

```
SQL> truncate table plan_table;

表被截断。
```

接下来分析一个 SQL 查询，查询用户 HR 中员工的部分信息，并使用 EXPLAIN 分析是否使用了索引。首先使用 HR 用户登录数据库，如实例 28-11 所示。

【实例 28-11】登录数据库。

```
SQL> connect hr/oracle@orcl
ERROR:
ORA-28002: the password will expire within 10 days


已连接。
```

下面使用 SQL 语句查询表 EMPLOYEES 中的 JOB 的种类，如实例 28-12 所示，使用 EXPLAIN PLAN FOR 分析该语句的执行。

【实例 28-12】使用 EXPLAIN 分析 SQL 语句的执行。

```
SQL> explain plan for
  2  select job_id,count(*)
  3  from employees
  4  group by job_id;
```

```
已解释。
```

接下来查询是否使用了分析结果，因为没有在 JOB_ID 上建立索引，所以语句的执行结果应该是使用全表扫描，查询结果如实例 28-13 所示。

【实例 28-13】查询 EXPLAIN 的分析结果。

```
SQL>select id,operation,options,object_name,position
  2* from plan_table

ID OPERATION          OPTIONS          OBJECT NAME     POSITION
--- ------------------------------- ---------------------------- ---
  0 SELECT STATEMENT                                          4
  1 HASH               GROUP BY                               1
  2 TABLE ACCESS         FULL           EMPLOYEES              1
```

从上述输出的第三行可以看出表访问使用全表扫描，因为 OPTIONS 值为 FULL，而 OBJECT_NAME 为 EMPLOYEES。下面创建基于列 JOB_ID 的索引，如实例 28-14 所示。

【实例 28-14】创建基于列 JOB 的索引。

```
SQL> create index emp_job_idx
  2  on employees(job_id);

索引已创建。
```

使用数据字典 USER_INDEXES 查询该索引的创建信息，如实例 28-15 所示。

【实例 28-15】查询索引 emp_job_idx 的创建信息。

```
SQL> col index_name for a20
SQL> col index_type for a15
SQL> col table_owner for a20
SQL> col table_name for a20
SQL> run
  1   select index_name,index_type,table_owner,table_name
  2  from user_indexes
  3* where index_name like 'EMP_JOB%'

INDEX_NAME      INDEX_TYPE      TABLE_OWNER      TABLE_NAME
--------------------------- ---------------------------- ---
EMP_JOB_IDX      NORMAL          HR                EMPLOYEES
```

显然索引 EMP_JOB_IDX 是基于列的索引。现在可以继续使用 EXPLAIN 执行对 SQL 语句的分析了。先截断表 PLAN_TABLE，该用户必须在 SYSTEM 用户下截断，如实例 28-16 所示。

【实例 28-16】截断表 PLAN_TABLE。

```
SQL> connect system/oracle@orcl
已连接。
SQL> truncate table plan_Table;
```

```
表被截断。
```

接下来，再次解析 SQL 语句的执行，如实例 28-17 所示。

【实例 28-17】使用 EXPLAIN 解析 SQL 语句。

```
SQL> connect hr/oracle@orcl
ERROR:
ORA-28002: the password will expire within 10 days


已连接。
SQL> explain plan for
  2  select job_id,count(*)
  3  from employees
  4  group by job_id;

已解释。
```

在建立了索引后，再次查询表 PLAN_TABLE 中记录的分析信息，查看是否使用了索引 EMP_JOB_IDX，如实例 28-18 所示。

【实例 28-18】在创建索引后继续查询 SQL 语句的执行计划。

```
SQL> select id,operation,options,object_name,position
  2  from plan_table;

ID OPERATION           OPTIONS         OBJECT_NAME   POSITION
--- -------------------------------- -------------------------- ------
0 SELECT STATEMENT                                          1
1 SORT                  GROUP BY NOSORT                     1
2 INDEX                 FULL SCAN           EMP_JOB_IDX      1
```

显然，从输入的第 3 行可以看出，该表使用了对表 EMPLOYEES 创建的索引 EMP_JOB_IDX。因为在 OPERATION 操作了索引 INDEX，而且对 OBJECT_NAME 值为 EMP_JOB_IDX 的索引进行了扫描操作（FULL SCAN）。

在数据库的分析以及设计中，如果需要复杂算法实现的查询，则最好创建基于该算法的函数，然后基于该函数对特定的列创建索引，在一定程度上会提高查询速度。下面以 SCOTT 用户 EMP 表为实例，说明创建基于函数的索引，如实例 28-19 所示。

【实例 28-19】创建基于函数的索引。

```
SQL> create index scott emp income idx
  2  on scott.emp(sal*12);

索引已创建。
```

在上例中，在 SCOTT 用户的表 EMP 上基于简单的计算公式创建了索引，索引名为 SCOTT_EMP_INCOME_IDX，在查询员工的年收入时，可以使用该索引减少查询时间。

使用数据字典 USER_INDEXES 查看是否成功创建索引，如实例 28-20 所示。

【实例 28-20】查看是否成功创建索引 SCOTT_EMP_INCOME_IDX。

```
SQL> col index_name for a20
SQL> col table_owner for a15
SQL> col table_name for a15
SQL> select index_name,index_type,table_owner,table_name
  2  from user_indexes
  3* where index_name like 'SCOTT%'

INDEX_NAME           INDEX_TYPE        TABLE_OWNER     TABLE_NAME
------------------------------------------------------------------
SCOTT_EMP_INCOME_IDX FUNCTION-BASED NORMAL  SCOTT          EMP
```

从输出可以看出索引 SCOTT_EMP_INCOME_IDX 记录在数据字典 USER_INDEX 中，属于用户 SCOTT 且基于表 EMP。

我们创建该索引的目的就是在查询时，希望 Oracle 使用索引查询提高响应速度，下面使用 EXPLAIN PLAN FOR 解析 SQL 语句，该 SQL 语句包含一个查询，查询表 EMP 中年薪少于 30000 的员工信息。

在使用 EXPLAIN PLAN FOR 之前，先截断表 PLAN_TABLE，如实例 28-21 所示。

【实例 28-21】截断表 PLAN_TABLE

```
SQL> truncate table plan_table;

表被截断。
```

然后，执行 EXPLAIN PLAN FOR 指令解析一个 SQL 语句，如实例 28-22 所示。

【实例 28-22】使用 EXPLAIN PLAN FOR 解析 SQL 语句。

```
SQL> explain plan for
  2  select ename,job,mgr,deptno
  3  from scott.emp
  4  where sal*12<30000;

已解释。
```

上述分析的 SQL 语句中，查询表 EMP 中员工年薪少于 30000 的信息，该语句的 WHERE 子句中包含条件 sal*12<30000，所以此时 Oracle 会使用基于函数的索引。下面查看表 PLAN_TABLE，判断上述 SQL 语句是否成功使用索引实现查询，如实例 28-23 所示。

【实例 28-23】查询 PLAN_TABLE。

```
SQL> select id,operation,options,object_name
  2  from plan_table;

ID OPERATION              OPTIONS             OBJECT_NAME
--- -------------------------------- -------------------------------
0 SELECT STATEMENT
1 TABLE ACCESS            BY INDEX ROWID       EMP
2 INDEX                   RANGE SCAN           SCOTT_EMP_INCOME_IDX
```

从查询结果的第 2 行（ID 为 1）可以看出，该查询使用了索引，通过索引中的行 ID（ROWID）查询需要的数据行，操作的对象为 EMP 表。


说 明

在使用 EXPLAIN PLAN FOR 指令时，SQL 语句并不真正执行，而只是分析 SQL 语句的执行。

在 Oracle 中也可以使用动态数据字典 v$sql_plan 查询一个 SQL 语句的执行计划，但是要求必须实际执行了这个 SQL 语句，如实例 28-24 所示。

【实例 28-24】执行对 EMP 表的查询语句。

```
SQL> select ename,job,mgr,deptno
  2  from scott.emp
  3  where sal*12<30000;

ENAME      JOB              MGR     DEPTNO
------------------ ------------------------- --------------
SMITH      CLERK             7902        20
JAMES      CLERK             7698        30
WARD       SALESMAN         7698        30
MARTIN     SALESMAN         7698        30
MILLER     CLERK             7782        10
TURNER     SALESMAN         7698        30
ALLEN      SALESMAN          7698        30
CLARK      MANAGER          7839        10

已选择 8 行。
```

接着，为了判断上述查询是否使用了索引 SCOTT_EMP_INCOME_IDX，可使用 v$sql_plan 来查询执行计划，如实例 28-25 所示。

【实例 28-25】使用 v$sql_plan 来查询执行计划。

```
SQL>  select id,operation,options,object_name,object_owner
  2   from v$sql_plan
  3  where object name like 'EMP%';

ID OPERATION       OPTIONS            OBJECT_NAME     OBJECT_OWNER
---------- -------------------- --------------------------- ----------
1 TABLE ACCESS     BY INDEX ROWID    EMP                 SCOTT
```

结果显示上例的查询 SQL 语句使用了索引，因为 OPTIONS 值为 BY INDEX ROWID。

在 Oracle 10g 之前的版本中，并不是默认使用基于函数的索引，所以需要设置一个参数 query_rewrite_enabled 为 true。而在 Oracle 10g 中，该参数默认值为 true，即在 Oracle 10g 中默认可以使用基于函数的索引，如实例 28-26 所示。

【实例 28-26】查询参数 query_rewrite_enabled 的值。

```
C:\Documents and Settings\Administrator>sqlplus /nolog
```

```
SQL*Plus: Release 10.2.0.1.0 - Production on 星期一 9月 28 20:31:46 2009

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> connect system/oracle@orcl
已连接。
SQL> show parameter query;

NAME                                 TYPE        VALUE
---------------------------------------------------------- ----
query_rewrite_enabled                        string      TRUE
query_rewrite_integrity                       string      enforced
```

上例是在 Oracle 10g 版本的数据库服务器上查询参数 query_rewrite_enabled 的值，显然该参数值为 TRUE，说明默认可以使用基于函数的索引。

下面为了更好地理解 AUTOTRACE 的使用，再使用 autotrace 来分析上述查询，如实例 28-27 所示。

【实例 28-27】使用 AUTOTRACE 分析 SQL 语句。

```
SQL> set autotrace traceonly
SQL> select ename,job,mgr,deptno
  2  from scott.emp
  3  where sal*12<30000;

已选择 8 行。


执行计划
----------------------------------------------------------
Plan hash value: 455855671

-------------------------------------------------------------------------

| Id  | Operation      | Name    | Rows  | Bytes | Cost  (%CPU)| Time     |
-------------------------------------------------------------------

|   0 | SELECT STATEMENT    |         |    8 |    416 |     2      (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP   | 8 |    416 |   2  (0)| 00:00:01 |
|*  2 |  INDEX RANGE SCAN | SCOTT_EMP_INCOME_IDX | 1 |  | 1  (0)| 00:00:01 |
---------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SAL"*12<30000)

Note
-----
   - dynamic sampling used for this statement
```

```
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        762  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          8  rows processed
```

读者注意在上例的输出中“加粗”的部分，在执行计划中的第二步中 OPERATION 为 TABLE ACCESS BY INDEX ROWID 说明该表访问使用索引，第三步扫描索引，通过索引值中 sal*12<30000 条件的记录找到相应的行 ID，通过 ROWID 找到所需要访问的行，这样不需要扫描表 EMP，而只需要通过索引，通过行 ID 直接检索表 EMP 中满足 sal*12<30000 条件的行。

统计信息中的最后一行“8 rows processed”，说明了此次查询检索的表的行数。该行数和 SQL 查询语句的检索结果是一致的，可以参考实例 28-24 的查询结果（已选择 8 行）。最后关闭自动追踪，如实例 28-28 所示。

【实例 28-28】关闭 AUTOTRACE 工具。

```
SQL> set autotrace off;
```

28.1.3 通过消除子查询优化 SQL 语句

使用 AUTOTRACE 查询存在子查询的 SQL 语句的执行结果。下面给出实例 28-29，对查询用户 SCOTT 的 EMP 表进行嵌套子查询，并分析该语句的执行过程。

【实例 28-29】对查询用户 SCOTT 的 EMP 表进行嵌套子查询。

```
SQL> select *
  2  from scott.emp e1
  3  where e1.sal>
  4  (select avg(sal)
  5   from scott.emp e2
  6   where e2.deptno = e1.deptno);
```

在上述 SQL 查询语句中，每扫描表 E1 中的一行，就执行一次子查询，这样如果表 E1 有 N 行，而自查也要执行 M 行，则每次执行需要执行 N*M 次操作。下面启动 AUTOTRACE 跟踪该 SQL 执行语句，如实例 28-30 所示。

【实例 28-30】开启 AUTOTRACE 功能。

```
SQL> set autotrace traceonly
```

下面跟踪 SQL 语句的执行，如实例 28-31 所示。

【实例 28-31】跟踪 SQL 语句的执行。

```
SQL> select *
  2  from scott.emp e1
  3  where e1.sal>
  4  (select avg(sal)
  5   from scott.emp e2
  6   where e2.deptno = e1.deptno);


执行计划
----------------------------------------------------------
Plan hash value: 2849664444

----------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost  (%CPU)| Time     |
----------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     2 |   174 |     6   (0)| 00:00:01 |
|*  1 |  FILTER            |      |       |       |            |          |
|   2 |   TABLE ACCESS FULL | EMP  |    12 |  1044 |     3   (0)| 00:00:01 |
|   3 |   SORT AGGREGATE   |      |     1 |    28 |            |          |
|*  4 |    TABLE ACCESS FULL| EMP  |     1 |    28 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("E1"."SAL"> (SELECT AVG("SAL") FROM "SCOTT"."EMP" "E2"
          WHERE "E2"."DEPTNO"=:B1))
   4 - filter("E2"."DEPTNO"=:B1)

Note
-----
   - dynamic sampling used for this statement


统计信息
----------------------------------------------------------
        53  recursive calls
         0  db block gets
        95  consistent gets
         0  physical reads
         0  redo size
      1007  bytes sent via SQL*Net to client
       385  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         3  sorts (memory)
         0  sorts (disk)
         5  rows processed
```

上述 SQL 语句的执行迭代次数很多，数据的一致性读次数也很多，如果随着表的增大会造成

比对的指数增长，会极大地消耗 CPU 资源。

下面改写 SQL 语句，即使用联机视图改写子查询，并继续使用 AUTOTRACE 分析，如实例 28-32 所示。

【实例 28-32】跟踪改写的 SQL 语句。

```
SQL> select *
  2  from emp e1,(select e2.deptno deptno,avg(e2.sal) avg_sal
  3             from scott.emp e2
  4             group by deptno) dept_avg_sal
  5  where e1.deptno = dept_avg_sal.deptno
  6* and e1.sal > dept_avg_sal.avg_sal


执行计划
----------------------------------------------------------
Plan hash value: 2230095667

---------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     2 |   228 |     8  (28)| 00:00:01 |
|*  1 |  HASH JOIN           |      |     2 |   228 |     8  (28)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | EMP  |    12 |  1044 |     3   (0)| 00:00:01 |
|   3 |   VIEW               |      |    12 |   312 |     4  (28)| 00:00:01 |
|   4 |    HASH GROUP BY     |      |    12 |   312 |     4  (28)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| EMP  |    12 |   312 |     3   (0)| 00:00:01 |
----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E1"."DEPTNO"="DEPT_AVG_SAL"."DEPTNO")
       filter("E1"."SAL">"DEPT_AVG_SAL"."AVG_SAL")

Note
-----
   - dynamic sampling used for this statement


统计信息
----------------------------------------------------------
         20  recursive calls
          0  db block gets
         62  consistent gets
          0  physical reads
          0  redo size
       1212  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
             4  sorts (memory)
             0  sorts (disk)
             5  rows processed
```

使用联机视图重新编写的 SQL，使得该 SQL 语句的查询过程只需要 N+M 次操作即可完成，显然其伸缩性增强，因为计算不会呈指数增长，显然从输出可以看出迭代的 CALLS 在减少，而且数据的一致性 GETS 也减少，随着表 EMP 的数据量的增加，修改后的查询会极大地减少系统 CPU 的消耗。

28.2　优化 SGA

Oracle 的 SGA 是指系统全局区，它是数据库运行期间使用的一段公有内存，即所有使用数据库的用户都可以访问这部分内存，它由共享池、重做日志缓冲区、数据库缓存高速缓冲区、Java 池、大池以及流池组成，如图 28-1 所示。其实优化 SGA 就是调整这些数据库组件的参数，如提高系统的运行效率、提高用户查询的响应事件等。

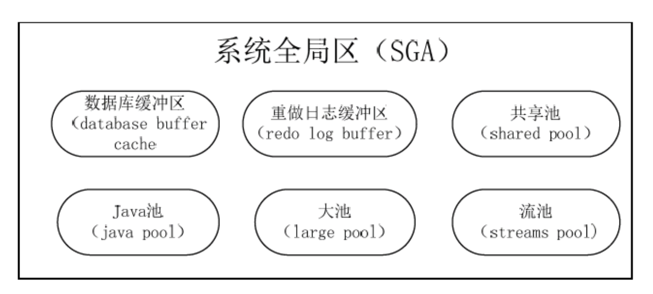


图 28-1　Oracle 的 SGA 组成图

这里简单解释一下上述各个组件的作用以及涉及的参数，这样读者在修改上述组件的尺寸时就更有针对性，做到“有的放矢”。

- 数据库缓冲区：该区域存放用户从数据库中读取的数据，在用户查找数据库时首先在数据库缓存中搜索，如果没有才会读取数据库文件，所以该区域不能设置的过小，不然频繁的读取数据文件会增大查询时间，因为磁盘 I/O 是耗时的行为。
- 重做日志缓冲区：该缓冲区放置用户改变的数据，所有变化了的数据和回滚需要的数据都暂时保存在重做日志缓冲区中。涉及的参数为 log_buffer，如实例 28-33 所示为查询重做日志缓冲区的大小。
- 共享池：共享池包括数据字典高速缓存和库高速缓存，库高速缓存用于存放 Oracle 解析的 SQL 语句、PL/SQL 过程、包以及各种控制结构，如锁、库缓冲句柄等。而数据字典高速缓存保存执行 SQL 语句所需的各种数据字典定义，如表和列的定义、用户访问表的权限等。
- Java 池：执行 Java 代码的区域。它为 Oracle 数据库中运行的 JVM（JAVA 虚拟机）分配一段固定大小的内存。
- 大池：该内存区提供大型的内存分配，在共享服务器连接模式下提供会话区，在使用 RMAN 备份时也使用该内存区作为磁盘 I/O 的数据缓冲区。

- 流池：该区域称为流内存，为 Oracle 流专用的内存池，流是 Oracle 数据库中的一个数据共享，其大小可以通过参数 stream_pool_size 动态调整。

【实例 28-33】查询重做日志缓冲区的大小。

```
SQL> show parameter log buffer;

NAME                                TYPE        VALUE
---------------------------------------------------------- -------
log_buffer                              integer      7024640
```

在 Oracle 11g 以及更高版本中，SGA 中的内存参数可以动态修改，但是总的内存大小受到参数 SGA_MAX_SIZE 的限制。在安装数据库时，这个参数的值是默认的，而实际的生产数据库往往需要重新设置一个新值，以利用操作系统中充足的内存资源。查看参数 SGA_MAX_SIZE 的值，如实例 28-34 所示。

【实例 28-34】查看参数 SGA_MAX_SIZE 的值。

```
SQL> show parameter sga_max_size;

NAME                                TYPE        VALUE
----------------------------------------------------------- ------
sga_max_size                          big integer    576M
```

继续查看 SGA 信息，如实例 28-35 所示。

【实例 28-35】查看 SGA 信息。

```
SQL> show sga;

Total System Global Area  603979776 bytes
Fixed Size                 1280380 bytes
Variable Size            222301108 bytes
Database Buffers          373293056 bytes
Redo Buffers               7135232 bytes
```

上述输出的第一个参数 Total System Global Area 其实和实例 28-11 中查到的一个数据大小相等，如下所示：

```
SQL> select 576*1024*1024 bytes
  2  from dual;

    BYTES
-----------------
 603979776
```

下面调整 SGA 的最大尺寸，目的是增大 Oracle 在整个内存中所占的比例，但是不能太大，一般可以设置为当前内存大小的一半即可。修改参数 SGA_MAX_SIZE 以修改 SGA 的尺寸，如实例 28-36 所示。

【实例 28-36】修改 SGA_MAX_SIZE 参数。

```
SQL> alter system set sga_max_size= 700M scope = spfile;

系统已更改。
```

在实例 28-36 中，把 SGA_MAX_SIZE 改为 700M，下面查询这次修改，如实例 28-37 所示。

【实例 28-37】查询参数 SGA_MAX_SIZE 修改结果。

```
SQL> show parameter sga_max_size;

NAME                                TYPE        VALUE
---------------------------------------------------------
sga_max_size                             big integer    576M
```

观察 VALUE 的值发现该值为 576M，没有修改，其实需要向读者说明的是参数 SGA_MAX_SIZE 是静态参数，需要重启数据库后方可生效。先不关闭数据库，继续对 SGA 进行优化。

查看在 Oracle 的静态参数中有哪些和 SGA 相关，如实例 28-38 所示。

【实例 28-38】查看和 SGA 相关的静态参数。

```
SQL> show parameter sga;

NAME                                TYPE        VALUE
--------------------------------------------------------- -----
lock sga                             boolean         FALSE
pre_page_sga                          boolean         FALSE
sga_max_size                          big integer      576M
sga_target                             big integer      576M
```

下面依次介绍参数 LOCK_SGA、PRE_PAGE_SGA 和 SGA_TARGET 对于优化 SGA 的作用。

1. LOCK_SGA 的含义及优化

该参数的作用是将 SGA 锁定（Lock）在物理内存内，这样就不会发生 SGA 使用虚拟内存的情况，显然这样可以提高数据的读取速度，记住磁盘 I/O 操作永远是尽量避免或减少的。该参数的默认值为 FALSE，即不将 SGA 锁定在内存中。下面修改参数 LOCK_SGA 为 TRUE，如实例 28-39 所示。

【实例 28-39】设置参数 LOCK_SGA 为 TRUE。

```
SQL> alter system set lock_sga = true scope = spfile;

系统已更改。
```

该参数是静态参数，需要重启数据库才可生效。

2. PRE_PAGE_SGA 的含义及优化

该参数的作用是启动数据库实例时，将整个 SGA 读入物理内存，对于内存充足的系统而言，这样显然可以提高系统运行效率。修改该参数为 TRUE，如实例 28-40 所示。

【实例 28-40】设置参数 PRE_PAGE_SGA 为 TRUE。

```
SQL> alter system set pre_page_sga= true scope = spfile;

系统已更改。
```

下面关闭数据库并重启数据库，如实例 28-41 所示。

【实例 28-41】关闭并重启数据库。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL> startup
ORA-32004: obsolete and/or deprecated parameter(s) specified
ORACLE 例程已经启动。

Total System Global Area  734003200 bytes
Fixed Size                  1281172 bytes
Variable Size             356518044 bytes
Database Buffers          369098752 bytes
Redo Buffers                7135232 bytes
数据库装载完毕。
数据库已经打开。
```

查看刚才修改的三个参数：SGA_MAX_SIZE、LOCK_SGA、PRE_PAGE_SGA，如实例 28-42 所示。

【实例 28-42】查看与 SGA 相关的参数修改结果。

```
SQL> show parameter sga;

NAME                                 TYPE        VALUE
------------------------------------------------------------
lock_sga                             boolean     TRUE
pre_page_sga                         boolean     TRUE
sga_max_size                         big integer 700M
sga_target                           big integer 576M
```

从上述输出可以看出参数 LOCK_SGA 和 PRE_PAGE_SGA 的值都为 TRUE，参数 SGA_MAX_SIZE 的值也修改为 700M。读者或许会问，参数 SGA_TARGET 是什么作用呢？

3. SGA_TARGET 的含义及优化

在 Oracle 10g 及以上的版本中，提供了内存的自动管理，这样 Oracle 可以根据业务需要和服务器自身的软硬件环境自动调整一些内存参数。参数 SGA_TARGET 就是决定是否使用 SGA 自动管理，该参数的默认值和系统的 SGA_MAX_SIZE 一样大，当该参数值不为 0 时，则启动 SGA 的自动管理，该参数可以动态修改，修改该参数的值为 700M，如实例 28-43 所示。

【实例 28-43】修改参数 SGA_TARGET 的值。

```
SQL> alter system set sga_target = 700M;

系统已更改。
```

SGA 可以自动管理，但不是所有的内存组件都可以自动调整，那么哪些 SGA 的内存是可以自动调整的呢？可以自动调整的内存组件如下。

- 共享池。
- Java 池。
- 大池。
- 数据库缓冲区。
- 流池。

这些组件的尺寸不需要用户干预，其值自动设置为 0，可使用视图 v$parameter 查看这些自动调整的内存组件的信息，如实例 28-44 所示。

【实例 28-44】查看自动调整的内存组件的信息。

```
SQL> col name for a30
SQL> col value for a20
SQL>select name,value,isdefault
  2  from v$parameter
  3  where name in ('shared pool size','large pool size',
  4* 'java_pool_size')

NAME                          VALUE               ISDEFAULT
------------------------------------------------- ---------
shared_pool_size                     0                 TRUE
large_pool_size                      0                 TRUE
java_pool_size                       0                 TRUE
```

虽然这些参数是可以自动调整的，但是用户依然可以使用 ALTER SYSTEM SET 指令修改内存组件的尺寸，如下例所示，修改 java_pool_size 的值为 10M。

```
SQL> alter system set java_pool_size = 10M;

系统已更改。
```

下面查询修改结果，如实例 28-45 所示。

【实例 28-45】查询参数 java_pool_size 的修改结果。

```
SQL> show parameter java_pool_size;

NAME                                TYPE        VALUE
------------------------------------------------------------ --------
java_pool_size                           big integer    12M
```

显然参数 java_pool_size 的值不再是 0，而 VALUE 值变为 12M。读者或许会问设置该参数值

为 10M，怎么变成 12M 呢？其实 Oracle 会针对系统自身情况做一些调整，是数据库自己的行为，读者不必太在意。

28.3 将程序常驻内存

在 Oracle 数据库中有一个软件包 DBMS_SHARED_POOL，它提供过程 KEEP 和 UNKEEP 将用户经常使用的程序，如存储过程、触发器、序列号、游标以及 JAVA SOURCE 等数据库对象长期保存在一个内存结构中，这个内存区就是共享池（Shared Pool），对于用户频繁使用的数据库对象而言，将其常驻内存可以减少磁盘 I/O，从而减少用户的响应时间。本节先讲解几个数据块缓冲池，分别解释它们的作用以及使用时间，然后介绍如何将一个存储过程常驻内存，最后介绍创建软件包 DBMS_SHARED_POOL 的 dbmspool.sql 过程，从而更清楚地理解软件包中各种过程的作用以及参数含义。

在 Oracle 数据库中，软件包 DBMS_SHARED_POOL 不是默认安装的，所以需要执行一个 SQL 脚本文件来创建该软件包，它有两个经常使用的过程 KEEP 和 UNKEEP，KEEP 过程将程序常驻内存，而 UNKEEP 将指定的程序清除出内存。首先使用 DBA 用户登录数据库，如实例 28-46 所示。

【实例 28-46】登录数据库并执行 KEEP 过程。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL> execute dbms_shared_pool.keep('HR.SECURE_DML');
BEGIN dbms shared pool.keep('HR.SECURE DML'); END;

      *
第 1 行出现错误:
ORA-06550: 第 1 行, 第 7 列:
PLS-00201: 必须声明标识符 'DBMS_SHARED_POOL.KEEP'
ORA-06550: 第 1 行, 第 7 列:
PL/SQL: Statement ignored
```

显然，提示执行失败，说明没有可用的软件包，需要手工创建该软件包。该软件包在笔者的计算机上位于目录 F:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN 下，脚本文件名为 dbmspool.sql。其实在这个目录下还有很多其他脚本文件，如熟悉的创建 SCOTT 用户的脚本文件 SCOTT.SQL。执行该脚本文件，如实例 28-47 所示。

【实例 28-47】创建 DBMS_SHARED_POOL 软件包。

```
F:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN

程序包已创建。


授权成功。


视图已创建。
```

```
程序包体已创建。
```

从输出可以看出，此时已成功创建软件包，并且在执行脚本文件 dbmspool.sql 的过程中，实现了授权和视图创建，并同时创建了过程 KEEP 和 UNKEEP（当然还有其他过程）。

如果用户在 SCOTT 用户或其他非 SYSTEM 用户下登录数据库并且尝试创建软件包 DBMS_SHARED_POOL，会提示出错，如下所示：

```
SQL> connect scott/tiger@orcl
已连接。
SQL> F:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN\dbmspool.sql;

程序包已创建。


授权成功。

  from dba_object_size
       *
第 4 行出现错误:
ORA-01031: 权限不足



警告: 创建的包体带有编译错误。
```

显然从输出可以看出当前用户缺少足够的权限，只要使用 SYSTEM 用户登录且赋予 DBA 角色即可。现在成功创建软件包 DBMS_SHARED_POOL 后就可以使用它的过程 KEEP 来将程序常驻内存了。

软件包 DBMS_SHARED_POOL 是过程的集合，包含常用的 KEEP 和 UNKEEP 过程，使用 KEEP 过程将用户频繁使用的程序常驻共享池中，使用 UNKEEP 将指定的程序从共享池中清除。

首先使用 SYSTEM 用户登录数据库：

```
SQL> connect system/oracle@orcl
已连接。
```

然后，通过数据字典 DBA_OBJECTS 查询用户 HR 的一个存储过程，假设该存储过程是用户程序频繁调用的过程，然后将其常驻内存，如实例 28-48 所示。

【实例 28-48】查看用户 HR 拥有的存储过程。

```
SQL> select object_name,object_type
  2  from dba_objects
  3  where object_type ='PROCEDURE'
  4  and owner ='HR';

OBJECT_NAME          OBJECT_TYPE
---------------------------------- ------------------------------
SECURE_DML           PROCEDURE
```

```
ADD_JOB_HISTORY       PROCEDURE
```

从查找结果可以看出，用户 HR 有两个存储过程，一个为 SECURE_DML，另一个为 ADD_JOB_HISTORY，将过程 SECURE_DML 常驻内存，或许读者想知道如何查看该过程的内容，毕竟对过程的功能了解得越多就更理解为什么将其常驻内存，如实例 28-49 所示。

【实例 28-49】查看过程 SECURE_DML 的内容。

```
SQL> SET LINE 100
SQL>select line,text
  2  from dba_source
  3* where name ='SECURE_DML'

     LINE TEXT
---------- ------------------------------------
        1 PROCEDURE secure_dml
        2 IS
        3 BEGIN
        4   IF TO_CHAR (SYSDATE, 'HH24:MI') NOT BETWEEN '08:00' AND '18:00'
        5        OR TO CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN') THEN
        6     RAISE_APPLICATION_ERROR (-20205,
        7            'You may only make changes during normal office hours');
        8   END IF;
        9 END secure_dml;

已选择 9 行。
```

该过程的作用很简单，就是判断某种状态下的系统时间如果不在 8~18 点之间或者日期为周六或周日就提示错误：You may only make changes during normal office hours。这里不再过多分析这个过程，读者只需要知道数据字典 DBA_SOURCE 的作用即可。下面演示如何将过程 SECURE_DML 常驻内存，如实例 28-50 所示。

【实例 28-50】将过程 SECURE_DML 常驻内存。

```
SQL> EXECUTE dbms_shared_pool.keep('HR.SECURE_DML');

PL/SQL 过程已成功完成。
```

输出提示已经成功创建 PL/SQL 过程，为了确认创建结果，可使用数据字典 V$db_object_cache 进行查看，它的作用是存储关于数据库对象在缓存中的信息，如实例 28-51 所示。

【实例 28-51】查看信息。

```
SQL> col name for a20
SQL> select name,namespace,sharable_mem,executions,kept
  2  from v$db_object_cache
  3* where owner ='HR'

NAME         NAMESPACE       SHARABLE_MEM EXECUTIONS  KEPT
-------------------- ------------------------------ -----------------------
SECURE_DML  TABLE/PROCEDURE    12837         0           YES
```

此时，输出说明用户 HR 的数据库对象，即存储过程 SECURE_DML，已经保存在共享池中，因为 KEPT 列的值为 YES。

既然可以使得一个程序常驻内存，同样有方法将其从内存清除，现在使用软件包 DBMS_SHARED_POOL 的 UNKEEP 过程将用户 HR 的过程 SECURE_DML 清除出内存，如实例 28-52 所示。

【实例 28-52】将用户 HR 的过程 SECURE_DML 清除出内存。

```
SQL> EXECUTE dbms_shared_pool.unkeep('HR.SECURE_DML');

PL/SQL 过程已成功完成。
```

在执行清除任务后，再次使用数据字典 v$db_object_cache 来查看清除结果，如实例 28-53 所示。

【实例 28-53】查看是否从内存清除过程 SECURE_DML。

```
SQL>  select name,namespace,sharable_mem,executions,kept
  2  from v$db_object_cache
  3  where owner ='HR';

NAME         NAMESPACE        SHARABLE_MEM EXECUTIONS KEP
-------------------- ------------------------------- ------
SECURE_DML  TABLE/PROCEDURE  12837          0          NO
```



说　明

如果将实现了常驻内存的程序从内存清除，则该程序的记录仍然在数据字典 v$db_object_cache 中，只是 KEPT 列的值为 NO。如果没有将一个程序常驻内存，则在数据字典 v$db_object_cache 中不存在该程序的任何记录。

上面我们使用软件包 DMBS_SHARED_POOL 将用户 HR 的一个过程 SECURE_DML 常驻内存，同时又使用了软件包过程 UNKEEP 将该过程从内存清除，那么软件包 DBMS_SHARED_POOL 到底是如何创建的呢？下面来分析脚本文件 dbmspool.sql，从而可以更清楚地理解软件包的作用，以及其中包含的过程含义。以下从脚本文件中截取部分内容进行详细说明。

```
create or replace package dbms shared pool is
  ------------
  --  OVERVIEW
  --
  --  This package provides access to the shared pool.  This is the
  --  shared memory area where cursors and PL/SQL objects are stored.
```

这部分说明该软件包的作用是提供对共享池的访问，使得游标或者 PL/SQL 对象（如存储过程、函数等）可以存储在共享池中。

该软件包中还包括 4 个函数，我们先介绍 KEEP 函数和 UNKEEP 函数。关于 KEEP 函数脚本中的内容如下所示。

```
procedure keep(name varchar2, flag char DEFAULT 'P');
  --  Keep an object in the shared pool.  Once an object has been keeped in
  --   the shared pool, it is not subject to aging out of the pool.  This
  --   may be useful for certain semi-frequently used large objects since
```

```
    --   when large objects are brought into the shared pool, a larger
    --   number of other objects (much more than the size of the object
    --   being brought in, may need to be aged out in order to create a
    --   contiguous area large enough.
    --   WARNING:  This procedure may not be supported in the future when
    --   and if automatic mechanisms are implemented to make this
    --   unnecessary.
```

该函数的作用就是将一个数据库对象常驻共享池，使得频繁访问的数据库对象（如大对象等）提高用户的访问响应时间，提高访问速度。该函数有两个参数。

- 第一个参数 name：该参数用来说明数据库对象的名字，这些数据库对象可以是 PL/SQL 过程、触发器、序列号以及 Java 对象。如果是 PL/SQL 过程可以使用“模式名.过程名”的方式指定特定模式的数据库过程名，如 scott.hispackage。
- 第二个参数 flag：该参数的作用是说明要常驻的数据库对象的类型，默认类型为包、过程或函数。否则需要使用一个字符变量说明对象类型。字符值与其代表的对象类型的对应关系如下所示。

```
Value       Kind of Object to keep
-----       ----------------------
P         package/procedure/function
 Q          sequence
  R          trigger
  T          type
  JS         java source
  JC         java class
  JR         java resource
  JD         java shared data
 C          cursor
```

关于 UNKEEP 过程的内容如下所示。

```
procedure unkeep(name varchar2, flag char DEFAULT 'P');
  -- Unkeep the named object.
  --   WARNING:  This procedure may not be supported in the future when
  --   and if automatic mechanisms are implemented to make this
  --   unnecessary.
  -- Input arguments:
  --   name
  --     The name of the object to unkeep.  See description of the name
  --     object for the 'keep' procedure.
  --flag
  --     See description of the flag parameter for the 'keep' procedure.
  -- Exceptions:
  --   An exception will raised if the named object cannot be found.
```

因为有了对于 KEEP 过程的详细说明，而过程 UNKEEP 的参数含义与 KEEP 过程的相同，所以不再重述 UNKEEP 过程的参数含义。

在该脚本文件中还有一条重要的授权语句，如下所示。

```
grant execute on dbms_shared_pool to execute_catalog_role
```

将对软件包 DBMS_SHARED_POOL 的执行权利赋予角色 execute_catalog_role，而当前的 SYSTEM 用户具有 DBA 权限，所以自动具有角色 execute_catalog_role，也可以通过数据字典查询当前用户具有的角色，如实例 28-54 所示。

【实例 28-54】查看当前用户的角色信息。

```
SQL> select *
  2  from dba_roles
  3  where role like 'EXECUTE%';

ROLE                          PASSWORD
-------------------------------------------- ------------------------
EXECUTE_CATALOG_ROLE             NO
```

上例说明了角色 execute_catalog_role 的存在，所以当前用户在创建了软件包 DBMS_SHARED_POOL 后就可以使用它了。希望读者在使用脚本文件时一定要仔细阅读脚本文件的内容，这样就可以从本质上理解一个软件包的作用和其中包含的其他过程了。

28.4 将数据常驻内存

在生产数据库中，为了提高用户的访问速度，对于经常使用的表，可以使其常驻内存中，这样就避免了对该表访问时产生频繁的磁盘 I/O 行为，这样可以提高用户的访问响应时间。而当不需要频繁访问该表时，DBA 可以将其从内存中清除。本节将继续学习 Oracle 的各种数据块的缓存池，通过分析了解将数据常驻内存的必要性和可行性，然后给出一个具体的实例，用于演示 SCOTT 用户的 EMP 表和一个索引常驻内存的过程。

在 Oracle 数据库体系结构的介绍中，读者已经知道在数据库块写到磁盘文件之前， 或者从磁盘文件读取数据之后，首先需要将数据块缓存在数据库高速缓存中，所以需要适当设置该缓冲区的大小以满足用户需求。在 Oracle 8i 之后的版本中，用户可以把 SGA 中段的已缓存块放在 3 个缓冲池中。

- 默认池（default pool）：所有的段都放在这个池中，即原先的缓冲区池，如果没有指定数据的缓存位置，默认将数据缓存在这个池中。
- 保持池（keep pool）：对于用户频繁访问的数据（如表或索引等数据库对象的数据块）可以放置在这个候选的缓冲区池中。放在默认池中的数据块，虽然可以频繁访问，但是这些段数据会老化而退出默认池，所以最好放置在保持池中，使得数据可以长久保存。
- 回收池（recycle pool）：对于随机访问的大段可以放在这个缓冲区池中，因为大的数据段会很快老化退出缓冲池，导致缓冲区的频繁刷新输出，所以需要将随机访问的大段放置在这个缓冲区池中。

在 Oracle 数据库中保持池和回收池都是用户管理的，即这两个池的大小需要手工配置，而默认池是自动管理的，在 SGA 中分配。查看保持池的大小信息如实例 28-55 所示。

【实例 28-55】查看保持池的大小信息。

```
SQL> show parameter keep

NAME                                 TYPE        VALUE
------------------------------------------------------------------------ --
buffer_pool_keep                             string
control_file_record_keep_time                       integer        7
db_keep_cache_size                           big integer      0
```

从输出可以看出，对于手动配置的缓冲池保持池的大小，对应的参数 db_keep_cache_size 的值为 0。

在没有设置保持池和回收池前，在数据库中只使用默认池作为数据块的缓冲池。可以通过实例 28-56 查询当前数据库所使用的数据库块的缓冲池。

【实例 28-56】查看当前库的数据块的缓冲池。

```
SQL> select id,name,block_size,buffers
  2  from v$buffer_pool;

    ID  NAME                  BLOCK_SIZE    BUFFERS
---------- ----------------------------------- -----------------------
     3  DEFAULT                 8192          47904
```

显然，当前数据库只用一个默认的数据块缓冲池，在手工设置保持池后才会显示保持池作为数据块缓冲池的信息。

在优化时，我们需要根据实际的需求，将用户经常使用的表或者索引放在保持池 keep pool 中，接下来我们介绍如何设置保持池的大小，以及将数据表以及索引常驻内存（保持池）中。

下面演示将用户 SCOTT 的 SALGRADE 表以及表 EMP 中建立的基于函数的索引 SCOTT_EMP_INCOME_IDX 常驻保持池中。

保持池的大小需要手工设置，显然这个尺寸是多少，应该基于常驻保持池中的数据大小，因为我们要将一个表以及索引保存在保持池中，所以需要先确认这些数据库对象的大小，如实例 28-57 所示。

【实例 28-57】查看表 SALGRADE 和索引 SCOTT_EMP_INCOME_IDX 的块大小。

```
SQL> col segment name for a20
SQL>select segment_name,segment_type,blocks
  2  from dba_segments
  3  where owner ='SCOTT'
  4* and segment name in ('SALGRADE','SCOTT EMP INCOME IDX')

SEGMENT_NAME          SEGMENT_TYPE           BLOCKS
-------------------------------------- -----
SALGRADE               TABLE                       8
SCOTT_EMP_INCOME_IDX  INDEX                        8
```

表 SALGRADE 和索引 SCOTT_EMP_INCOME_IDX 的大小都为 8 个数据库块大小。读者需要注意数据字典 dba_segments 是静态数据字典，如果需要获得最新的段统计信息，需要使用

ANALYZE 指令收集统计信息，如实例 28-58 所示。

【实例 28-58】收集表和索引的最新统计信息。

```
SQL> analyze index scott.SCOTT_EMP_INCOME_IDX compute statistics;

索引已分析

SQL> analyze table scott.salgrade compute statistics;

表已分析。
```

那么在确认了表和索引占用的数据块数后，那么数据库块大小是多少呢？可通过实例 28-59 查询数据库的块大小。

【实例 28-59】查询数据库的块大小。

```
SQL> show parameter db_block_size;

NAME                                TYPE        VALUE
----------------------------------------------------------
db_block_size                               integer        8192
```

从输出可以看出，当前数据库的数据块大小为 8k，所以通过这些数据（索引和表占用的数据块数和数据块大小）可以计算当前将表和索引常驻内存需要多大，如实例 28-60 所示。

【实例 28-60】计算要保存的表和索引的大小。

```
SQL> select (8+8)*8 Kbytes from dual;

   KBYTES
----------
       128
```

我们需要 128k 的保持池大小，在确认了要保存的数据大小后，就可以手工设置保持池的大小，如实例 28-61 所示。

【实例 28-61】设置保持池的大小。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL>  alter system set db keep cache size = 10M;

系统已更改。
```

将保持池的大小设置为 10M，这样可以充分满足我们要存储的包 SALGRADE 和索引 SCOTT_EMP_INCOME_IDX 大小。接着查看当前数据库中数据块的缓冲池信息，如实例 28-62 所示。

【实例 28-62】查询当前库的数据块的缓冲池信息。

```
SQL> select id,name,block size,buffers
  2  from v$buffer_pool;

        ID NAME                 BLOCK_SIZE    BUFFERS
```

```
---------- -------------------- ---------- ----------
         1 KEEP                     8192       1497
         3 DEFAULT                  8192      46407
```

从输出可以看出，多了一个缓冲池 KEEP，该池的数据块大小为 8k，缓冲区大小为 1497 个数据库块大小，即 10M。

下面就可以将索引和数据表设置为常驻保持池中了，在设置之前，先看看表 SALGRADE 和索引 SCOTT_EMP_INCOME_IDX 当前存放在什么缓冲池中，如实例 28-63 所示。

【实例 28-63】查看表 SALGRADE 当前存放在什么缓冲池中。

```
SQL> connect scott/tiger@orcl
已连接。
SQL> select table_name,tablespace_name,buffer_pool
  2  from user_tables
  3  where table_name ='SALGRADE';

TABLE_NAME           TABLESPACE_NAME BUFFER_POOL
------------------------------------ ----------------------------
SALGRADE                USERS           DEFAULT
```

输出结果显示当前的表 SALGRADE 放在默认缓冲池中，因为 BUFFER_POOL 的值为 DEFAULT。下面再查看索引 SCOTT_EMP_INCOME_IDX 的缓存池，如实例 28-64 所示。

【实例 28-64】查看索引 SCOTT_EMP_INCOME_IDX 的缓存池。

```
SQL> select index_name,table_name,buffer_pool
  2  from user_indexes
  3  where index_name ='SCOTT_EMP_INCOME_IDX';

INDEX_NAME                    TABLE_NAME         BUFFER_POOL
-------------------------------------------------- ----------
SCOTT_EMP_INCOME_IDX            EMP                DEFAULT
```

输出显示当前的索引 SCOTT_EMP_INCOME_IDX 放在默认缓冲池中，因为 BUFFER_POOL 的值也为 DEFAULT。下面将表和索引分别设置为常驻保持池中，如实例 28-65 所示。

【实例 28-65】将表 SALGRADE 设置为常驻内存。

```
SQL> alter table salgrade
  2  storage (buffer_pool keep);

表已更改。
```

现在通过数据字典 USER_TABLES 查看表 SALGRADE 的缓冲池信息，如实例 28-66 所示。

【实例 28-66】查看表 SALGRADE 的缓冲池。

```
SQL> select table_name,tablespace_name,buffer_pool
  2  from user_tables
  3  where table name ='SALGRADE';

TABLE_NAME           TABLESPACE_NAME BUFFER_POOL
```

```
------------------------------------ -----------------------------
SALGRADE                  USERS                  KEEP
```

从列 BUFFER_POOL 的值为 KEEP 可以知道，表 SALGRADE 已经设置为常驻内存（保持池）中了。下面设置索引常驻内存，如实例 28-67 所示。

【实例 28-67】将索引 scott_emp_income_idx 设置为常驻内存。

```
SQL> alter index scott_emp_income_idx
  2  storage(buffer_pool keep);

索引已更改。
```

现在使用数据字典 USER_INDEXES 查看对索引 SCOTT_EMP_INCOME_IDX 的缓冲池的修改结果，如实例 28-68 所示。

【实例 28-68】查看索引 SCOTT_EMP_INCOME_IDX 的缓冲池。

```
SQL> select index_name,table_name,buffer_pool
  2  from user_indexes
  3  where index_name = 'SCOTT_EMP_INCOME_IDX';

INDEX_NAME                      TABLE_NAME           BUFFER_POOL
-------------------------------------------------- ------
SCOTT_EMP_INCOME_IDX             EMP                  KEEP
```

显然，从列 BUFFER_POOL 的值为 KEEP 可以知道，索引 SCOTT_EMP_INCOME_IDX 已经设置为常驻内存了。

上面已将用户 SCOTT 的 SALGRADE 表以及表 EMP 中建立的基于函数的索引 SCOTT_EMP_INCOME_IDX 常驻保持池中。在不需要频繁访问这些表或索引时，可以将其恢复为默认缓冲池，这样就可以释放一部分内存，留给其他频繁访问的数据使用。下面先演示如何将表 SALGRADE 恢复为默认缓冲池，如实例 28-69 所示。

【实例 28-69】将表 SALGRADE 恢复为默认缓冲池。

```
SQL> alter table salgrade
  2  storage(buffer_pool default);

表已更改。
```

接着可以查看修改结果，确认是否将表 SALGRADE 的缓冲池设置为默认缓冲池，如实例 28-70 所示。

【实例 28-70】查看表 SALGRADE 的缓冲池信息。

```
SQL> select table_name,tablespace_name,buffer_pool
  2  from user_tables
  3  where table_name ='SALGRADE';

TABLE_NAME          TABLESPACE_NAME BUFFER_POOL
------------------------------------ ------------------------------
SALGRADE                 USERS                DEFAULT
```

输出说明已经将表 SALGRADE 常驻内存改为使用默认缓冲区，因为 BUFFER_POOL 的值已经为 DEFAULT，以后对表 SALGRADE 的访问将把表数据读入默认缓冲区。

接下来将索引 SCOTT_EMP_INCOME_IDX 从常驻内存改为使用默认缓冲池，如实例 28-71 所示。

【实例 28-71】将索引 SCOTT_EMP_INCOME_IDX 恢复为默认缓冲池。

```
SQL> alter index scott_emp_income_idx
  2  storage (buffer_pool default);

索引已更改。
```

其实，与设置为常驻内存不同的是 STORAGE 子句中的一个参数，将设置常驻内存的 KEEP 参数改为 DEFAULT 就修改了索引的缓冲池设置。此时，已成功将索引 SCOTT_EMP_INCOME_IDX 的缓冲池设置为默认缓冲池。

显然此时，保持池依然占用内存，但是其中已经没有了数据，那么如何释放保持池中的内存呢？可使用 ALTER SYSTEM 指令来回收这段内存，如实例 28-72 所示。

【实例 28-72】会收保持池中的内存。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL> alter system set db_keep_cache_size = 0;

系统已更改。
```

此时，我们不再使用保持池作为缓冲池，可以使用数据字典 v$buffer_pool 来验证，如实例 28-73 所示。

【实例 28-73】查看与数据库相关的缓冲池信息。

```
SQL> select id,name,block_size,buffers
  2  from v$buffer_pool;

        ID NAME                 BLOCK_SIZE    BUFFERS
---------- -------------------------------- ---------------------
         3 DEFAULT                    8192      47904
```

显然此时只有默认缓冲池可以使用，说明保持池已经不再有效。

28.5　优化重做日志缓冲区

重做日志缓冲区是一段临时存储重做数据的内存区，用户所有变化了的原数据和修改后的数据都保存在重做日志缓冲区中，并由 LGWR 进程负责写入重做日志文件。在优化时，需要考虑该内存区的大小，以及 LGWR 的写速度和重做日志文件所在磁盘的争用等。本节首先讲述重做日志文件的工作机制、与重做日志相关的等待事件，最后给出相应地解决问题的思路，达到优化重做日志缓冲区的目的。

28.5.1 重做日志缓冲区的工作机制

在 SGA 中重做日志缓冲区一般是最小的一个内存结构，在用户对数据库做更改时，重做日志缓冲区为所有修改数据的服务器进程共享使用，这些服务器进程负责将更改数据的原始值和修改后的新值以及事务 ID 写入重做日志缓冲区，而 LGWR 进程负责将重做日志缓冲区中的数据写入重做日志文件，Oracle 的重做日志组是循环使用的，当覆盖以前的重做日志文件时，如果数据库处于归档模式则自动启动归档进程，ARCH 负责将被覆盖的重做日志文件的内容拷贝到归档日志文件，如图 28-2 所示为以上描述的行为的示意图。

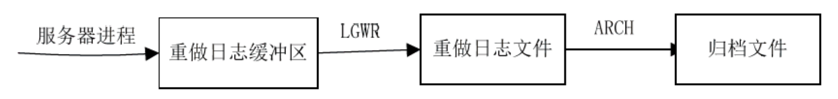


图 28-2 重做日志缓冲区以及相关进程的工作示意图

上图是一个静态图，其实在数据库内部上述活动是个十分活跃的行为，服务器进程修改数据库中的数据或表结构，不断地将相关的重做数据写入重做日志缓冲区，而重做日志缓冲区在一定的条件下，如每 3 秒钟，将其中的重做数据写入重做日志文件，而当重做日志文件切换时（无论是用户主动切换，还是数据库自己的行为）将导致归档进程 ARCH 启动，把重做日志文件中的数据读入归档文件，然后数据库才可以继续使用相关的重做日志文件，在这个过程中无论哪里出现问题都会导致一些等待事件，如果是频繁发生的等待事件，就会影响系统的性能，如重做日志缓冲区太小，而服务器进程写入速度又比 LGWR 写出的速度快就会出现 log buffer space 等待时间，此时就需要 DBA 主动采取优化了。

为了更清楚地理解在用户修改一行数据库时，与重做日志相关的一系列行为，下面给出一个过程分析。

01 用户发出一条更新的 SQL 语句，该语句是某个事务的一部分，Oracle 为该事务分配了唯一的事务号。

02 服务器进程负责将需要的数据、索引和还原数据读入内存，并将要更新的行加锁。

03 服务器进程获得重做拷贝闩锁（闩锁实现对重做日志缓冲区的串行使用），该闩锁是服务器进程访问重做日志缓冲区的第一步。此时如果没有其他的闩锁可用，则别的服务器进程无法使用重做日志缓冲区。

04 服务器进程获得重做分配闩锁，从而获得在重做日志缓冲区中的预留空间，此时释放重做分配闩锁。

05 服务器进程利用重做拷贝闩锁把重做项（更新数据的原始值、操作类型、事务号等信息）写入重做日志缓冲区，然后释放重做日志拷贝闩锁。

06 服务器进程把还原信息写入与该事务相关的还原段，还原段在用户使用 ROLLBACK 指令时使用。

07 服务器进程更新锁住的数据，将回滚所需的原始值和对数据所做的修改都写入数据库高速缓冲区，然后数据库高速缓冲区中的这些数据被标记为脏数据，因为目前内存和外存中的数据不一致。

在深入地了解了重做日志缓冲区的工作机制和过程后，下面分析 LGWR 进程何时将重做日志缓冲区的重做数据写入重做日志文件，理解这些内容对于优化重做日志缓冲区是很有必要的。

LGWR 把重做日志缓冲区写入重做日志文件的条件如下。

- 每隔 3 秒钟。
- 事务被提交时。
- 当重做日志缓冲区的记录的变化数据量超过 1MB。
- 当重做数据的大小为重做日志缓冲区大小的 1/3 时。这里需要说明，并不是重做日志缓冲区永远不会填到超过其 1/3 容量，而是说明当重做数据量达到其容量的 1/3 这个阀值时，LGWR 进程会写出重做日志缓冲区中的数据，而剩下的 2/3 的数据可以供其他服务器进程使用。
- 检验点发生时。
- 当 DBWR 进程将数据库高速缓冲区中的数据写到数据文件前。

下面查看重做日志缓冲区的尺寸，如实例 28-74 所示。

【实例 28-74】查看重做日志缓冲区的尺寸。

```
SQL> show parameter log_buffer;

NAME                                 TYPE        VALUE
------------------------------------------------------------
log_buffer                            integer     7024640
```

28.5.2 重做日志缓冲区的相关等待事件

如果需要优化重做日志缓冲区，必须首先确认发生了与重做日志缓冲区相关的等待事件，否则不应该随便调整重做日志缓冲区的尺寸。读者可以使用等待（WAIT）视图和事件（EVENT）视图，确认等待事件以及该事件涉及的文件和会话，如实例 28-75 所示。

【实例 28-75】通过数据字典视图查看会话等待事件

```
SQL> col event for a35
SQL> col username for a10
SQL> select sw.sid,s.username,sw.event,sw.wait time
  2 from v$session s,v$session_wait sw
  3 where sw.event not like 'rdbms%'
  4 and sw.sid = s.sid
  5* order by sw.wait time,sw.event

      SID USERNAME   EVENT                                WAIT_TIME
---------- ---------- ----------------------------------- ----------
      154 SYSTEM     SQL*Net message to client                   -1
      149           Streams AQ: qmn coordinator idle wa          0
                   it

      144           Streams AQ: qmn slave idle wait             0
      147           Streams AQ: waiting for time manage          0
                   ment or cleanup tasks
```

```
        159          jobq slave wait                         0
        170          pmon timer                             0
        164          smon timer                             0

已选择 7 行。
```


说 明

虽然出现了等待事件，但是该等待事件没有影响系统使用或系统性能，此时就不要轻易去优化。

下面分析和重做日志缓冲区相关的等待事件，以及事件发生的原因分析，一旦找到等待事件，并知道该事件发生的相关原因，就可以进行优化工作了。

- Log buffer space：该事件说明缺少重做日志的缓冲区空间，造成该等待事件的原因一般是服务器进程写入重做日志缓冲区的速度高于 LGWR 将重做日志缓冲区写出的速度，也有可能是重做日志文件所在磁盘设备速度慢或者存在设备争用，造成 LGWR 进程无法及时将重做日志缓冲区中的重做数据写入重做日志文件。优化方法：调整重做日志缓冲区尺寸，或者将重做日志数据文件迁移到高速磁盘上，或者为了解决争用，将重做日志文件、数据库数据文件以及归档文件放在不同的磁盘上。
- Log file parallel write：该事件的含义是日志文件并行写等待，是在将重做日志缓冲区中的重做数据写入磁盘引起的等待事件。造成该事件的原因一般是联机重做日志文件所在的设备速度慢或者存在磁盘争用。优化方法：将重做日志文件、数据库数据文件以及归档文件放在不同的磁盘上，并将重做日志文件放置在高速盘上。
- Log file single write：该等待事件仅与写日志文件头块有关，表示检查点中的等待。
- Log file switch（archiving needed）：该等待事件的含义是日志文件切换等待，对于处于归档模式的数据库而言，当日志组写满后，在日志切换时，如果需要覆盖先前的日志，而该日志需要归档进程写入归档文件，由于写入归档文件需要时间，而 LGWR 进程需要将重做日志缓冲区中的数据写入重做日志文件，而归档未完成需要等待，在此期间就产生了 Log file switch 事件，该等待事件的原因一般是 I/O 问题、ARCH 归档进程跟不上 LGWR 日志写进程的速度或者日志组太少引起的。优化方法：启用多个归档 ARCH 进程或 I/O 从进程（slave process），将归档的文件、数据文件或重做日志文件放置在不同的磁盘上，减少磁盘争用以减少 ARCH 归档进程的归档事件，或者增加重做日志组。
- Log file switch（checkpoint incomplete）：该事件是由于日志切换太频繁引起的，由于频繁地切换重做日志文件，造成检验点的排队。发生该等待事件的原因一般是重做日志缓冲区空间太小或者重做日志组太少。优化方法：增加重做日志组或者增加重做日志缓冲区的尺寸。
- Log file sync：当用户提交时，重做日志缓冲区中的数据会一次全部写到重做日志文件中，此时发生的 LGWR 的写出等待就是 log file sync 等待。造成该等待的原因一般是放置联机重做日志文件的磁盘存在争用或者磁盘速度慢。优化方法：将重做日志文件、数据文件或归档重做日志文件放置在不同的磁盘上，以减少数据库中的各种文件之间的 I/O 争用，同时可以把重做日志文件放在高速磁盘上，以减少将重做数据写入重做日志文件的时间。

- Latch free：该等待事件的含义是当前的服务器进程需要某个闩锁，如等待共享池的库高速缓存闩锁。如果发生该等待事件也可以通过数据字典 v$latch 查看相关的闩锁命中率，如实例 28-76 所示。

【实例 28-76】查询与闩锁 LATCH 相关的信息。

```
SQL> select latch#,name,gets,misses,1-(misses/(gets+misses)) "gets rate"
  2  from v$latch
  3  where misses>1;

   LATCH# NAME                          GETS     MISSES  gets rate
---------- -------------------------------------------------- ---------- -
      180 dummy allocation              1605      2      .998755445
      121 checkpoint queue latch         24231     9      .999628713
      213 shared pool                   96480     112    .998840484
    ..................
      215 library cache lock             45617    3      .999934239
      199 row cache objects              80118    9      .999887678
      214 library cache                  135162   59     .999563677
      146 redo writing                   6541     115    .982722356
      216 library cache pin              72865    3      .99995883
      19 enqueue hash chains             69022    5      .999927565
      302 session state list latch         1602     5     .996888612

已选择 18 行。
```

可以通过以上所示的 get rate 来进一步确认闩锁的命中率。

28.5.3 设置重做日志缓冲区的大小

在发生与重做日志缓冲区相关的等待事件时，或者在 DML 操作频繁的生产数据库中，往往需要增加重做日志缓冲区的尺寸。下面介绍如何修改重做日志缓冲区的大小并使其生效，如实例 28-77 所示。

【实例 28-77】查看重做日志缓冲区的大小。

```
SQL> show parameter log_buffer;

NAME                                TYPE        VALUE
-------------------------------------------------------------
log_buffer                               integer     7024640
```

从输出可以看出该重做日志缓冲区的大小为 7024640 字节，所以在设置该参数的值时，也必须利用事件字节数来修改该参数，为了更容易理解，可将字节数据转换成 M 字节的形式，并使用数据字典视图 v$parameter 查看，如实例 28-78 所示。

【实例 28-78】使用数据字典视图 v$parameter 查看重做缓冲区的大小。

```
SQL> col name for a20
SQL> select name,value/(1024*1024) "M字节"
```

```
  2  from v$parameter
  3* where name ='log_buffer'

NAME                          M 字节
------------------------------------ ----------
log_buffer                      6.69921875
```

从输出可以看到，当前数据库的重做日志缓冲区的尺寸为 7M。由于参数 LOG_BUFFER 是静态参数，所以设置该参数后必须重新启动数据库才可以生效，下面增大重做日志缓冲区的尺寸，设置为 10M。但是需要将 10M 转换成字节（1024*1024=10485760 字节），如实例 28-79 所示。

【实例 28-79】设置重做日志缓冲区大小为 10M。

```
SQL> alter system set log_buffer = 10485760 scope = spfile;

系统已更改。
```

为了使得设置的重做日志缓冲区参数生效，必须关闭数据库后重新启动，如实例 28-80 所示。

【实例 28-80】关闭并重启数据库。

```
SQL> connect system/oracle@orcl as sysdba
已连接。
SQL> shutdown immediate;
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL> startup
ORA-32004: obsolete and/or deprecated parameter(s) specified
ORACLE 例程已经启动。

Total System Global Area  734003200 bytes
Fixed Size                  1281172 bytes
Variable Size             197134492 bytes
Database Buffers          524288000 bytes
Redo Buffers               11329536 bytes
数据库装载完毕。
数据库已经打开。
```

重新打开数据库后，修改过的重做日志缓冲区的大小即可生效。查询重做日志缓冲区的修改结果，如实例 28-81 所示。

【实例 28-81】查询重做日志缓冲区的尺寸。

```
SQL> show parameter log_buffer;

NAME                                 TYPE        VALUE
-----------------------------------------------------------
log_buffer                           integer     11154432
```

我们看到此时 LOG_BUFFER 的 VALUE 为 11154432 字节，已经不是以前的 7024640 字节，说明在实例 28-80 中的修改已生效。

注意：设置的该缓冲区的大小为 10M，但是这里的数值为 10.6376953M，这是因为 Oracle 会根据系统情况做一些调整。

28.6　优化 PGA 内存

PGA 是程序全局区，该区域在数据库会话专有连接模式下是私有区域，服务器进程和用户进程一一对应，用户进程单独使用 PGA，在共享服务器会话连接模式下，一个服务器进程对应多个用户进程，PGA 是多个用户进程共享使用。PGA 主要用作大规模的数据排序，如用户输入的 SQL 语句有 GROUP BY 或 ORDER BY 子句等操作。

所谓的 PGA 优化就是将这些大规模的数据排序放在 PGA 中运行，而不是使用虚拟内存而占用操作系统的交换区（SWAP AREA）。这样就要求合理地设置 PGA 大小，从而使得排序区符合系统需要，在 Oracle 9i 之前，该排序区由参数 SORT_AREA_SIZE 决定，通过手工设置该参数，反复调整以满足系统需要。可以通过如实例 28-82 所示的方式查看该参数的值。

【实例 28-82】查看 PGA 中排序区的大小。

```
NAME                                TYPE        VALUE
--------------------------------------------------------- ---
sort_area_size                             integer       65536
```

如果出于大规模排序的需要，可以调整该参数为更大的值，但是问题是到底多少是最好的呢？显然如果分配过多，会造成系统的其他组件的内存不足，如果过少又不能满足排序的需要。在 Oracle 9i 及以上版本中（Oracle 10g、Oracle 11g）支持 PGA 排序区的自动调整功能。但是该自动调整有一个限制就是必须给出一个排序区的值，排序区会根据排序需要而在这个值内自动调整。该值由参数 PGA_AGGREGATE_TARGET 决定，同时为了启动 PGA 排序区的自动管理必须设置参数 WORKAREA_SIZE_POLICY 为 AUTO，这也说明要设置 PGA 排序区的自动管理必须配置两个参数，即 PGA_AGGREGATE_TARGET 和 WORKAREA_SIZE_POLICY。下面查看系统上这两个参数的值，如实例 28-83 所示。

【实例 28-83】查看 PGA 的排序区是否为自动管理。

```
SQL> col name for a20
SQL> col value for a30
SQL> select name,value,isdefault
  2  from v$parameter
  3* where name in ('pga_aggregate_target','workarea_size_policy')

NAME                 VALUE                          ISDEFAULT
-------------------------------- -------------------- ----------------------
pga_aggregate_target     200278016                          FALSE
workarea_size_policy     AUTO                           TRUE
```

从实例 28-83 的输出可以看出参数 WORKAREA_SIZE_POLICY 是默认的参数，因为其 ISDEFAULT 的值为 TRUE，而参数 PGA_AGGREGATE_TARGET 是需要设置的，它不是系统的

默认参数，因为 ISDEFAULT 的值为 FALSE。

下面分析一下当前数据库的 PGA 状态，如实例 28-84 所示。

【实例 28-84】通过数据字典视图 v$pgastat 查询 PGA 状态信息。

```
SQL> col name for a40
SQL> run
  1  select *
  2* from v$pgastat

NAME                                    VALUE      UNIT
------------------------------------------------------------------------
aggregate PGA target parameter          200278016    bytes
aggregate PGA auto target               162897888    bytes
global memory bound                     40054784    bytes
total PGA inuse                         19604480    bytes
total PGA allocated                     42294272     bytes
maximum PGA allocated                   90947584    bytes
total freeable PGA memory                      0     bytes
process count                                21
max processes count                          33
PGA memory freed back to OS                    0    bytes
total PGA used for auto workareas              0      bytes

NAME                                              VALUE UNIT
------------------------------------------------------------------
maximum PGA used for auto workareas        3831808   bytes
total PGA used for manual workareas              0    bytes
maximum PGA used for manual workareas            0  bytes
over allocation count                            0
bytes processed                          101315584  bytes
extra bytes read/written                         0   bytes
cache hit percentage                           100  percent
recompute count (total)                        869

已选择 19 行。
```

需要注意的是，以上输出中用于记录 PGA 状态的三个参数，即 aggregate PGA target parameter、aggregate PGA auto target 和 cache hit percentage，其含义如下所示。

- aggregate PGA target parameter：用户设置的当前 PGA 内存总和。
- aggregate PGA auto target：Oracle 为 PGA 的排序区分配的内存大小，显然其值不能超过 PGA 内存的总和。
- cache hit percentage：说明排序区在 PGA 的排序区完成的比例，100 percent 表示全部排序都在 PGA 的排序区内完成，所以 Oracle 自动分配的排序区的尺寸是合理的。

可以使用以下方式监控 PGA 的排序区是否合理，其目的是观察参数 cache hit percentage 的值，如果是 100 percent，则认为 Oracle 根据系统状态设置的 PGA 的排序区是合理的，否则就需要增加参数 PGA_AGGREGATE_TARGET 的值，以增加为 PGA 的排序区添加内存的需要，如实例 28-85

所示。

【实例 28-85】查看在 PGA 的排序区进行排序的百分比。

```
SQL> select *
  2  from v$pgastat
  3  where name like 'cache%';

NAME                                           VALUE  UNIT
----------------------------------------------------------------------
cache hit percentage                                   100     percent
```

显然其中的 VALUE 值为 100，所以当前的排序行为都在 PGA 的排序区内完成。如果出现部分排序不在 PGA 的排序区完成，即 VALUE 的值小于 100，则需要考虑适当增加 PGA 的内存总和。如实例 28-86 所示，通过设置参数 PGA_AGGREGATE_TARGET 来调整 PGA 的内存总和。

【实例 28-86】调整 PGA 的内存大小 。

```
SQL> alter system set pga aggregate target = 286M;

系统已更改。
```

查看修改结果，如实例 28-87 所示。

【实例 28-87】查看参数 PGA_AGGREGATE_TARGET 的修改结果。

```
SQL> show parameter pga

NAME                                TYPE        VALUE
---------------------------------------------------------- ---
pga_aggregate_target                         big integer    286M
```

再次查看此时 PGA 的状态信息，并查看 Oracle 为 PGA 的排序区分配了多少内存，如实例 28-88 所示。

【实例 28-88】查看 PGA 的大小以及 PGA 中排序区的大小。

```
SQL> select *
  2  from v$pgastat
  3  where name in ('aggregate PGA target parameter',
  4                'aggregate PGA auto target');

NAME                                           VALUE      UNIT
----------------------------------------------------------------------
aggregate PGA target parameter                      288435456    bytes
aggregate PGA auto target                          223792128    bytes
```

我们看到此时的 PGA 总和为 286M，同时自动分配给 PGA 的排序区的尺寸为 213.424805M，可见随着 PGA 的总内存的增加，Oracle 会自动增加其为排序区分配的内存容量。

读者也可以通过数据字典视图 v$pga_target_advice 获得 PGA 的排序区进行排序行为的更为详细的信息：

```
SQL> select pga_target_for_estimate/(1024*1024) as estd_target,
  2  estd_pga_cache_hit_percentage ,estd_overalloc_count
  3  from v$pga_target_advice;

ESTD_TARGET ESTD_PGA_CACHE_HIT_PERCENTAGE ESTD_OVERALLOC_COUNT
--------------------- ------------------------ --------------------
        32                          100                   0
        64                          100                   0
       128                          100                   0
       192                          100                   0
       286                          100                   0
 307.199219                           100                   0
 358.399414                           100                   0
 409.599609                           100                   0
 460.799805                          100                  0
       512                         100                  0
       768                         100                  0

ESTD_TARGET ESTD_PGA_CACHE_HIT_PERCENTAGE ESTD_OVERALLOC_COUNT
----------------- ---------------------------------------------------
      1024                          100                  0
      1536                          100                  0
      2048                          100                  0

已选择 14 行。
```

从以上输出可以看出，Oracle 给出了一系列对 PGA 的估计值，并且给出了每个估计值状态下，排序在 PGA 的排序区完成的比率，由参数 ESTD_PGA_CACHE_HIT_PERCENTAGE 决定。由于笔者的计算机上没有任何排序行为，所以对于 PGA 的每一个估计值的所有排序都在 PGA 的内存中完成。如果在生产数据库系统上，参数 ESTD_PGA_CACHE_HIT_PERCENTAGE 一般不会都是 100，可以根据这个参数的提示来确定手动设置的 PGA 内存总和是否合适，合适的含义是要求所有的排序行为都在 PGA 的内存中完成，这样通过内存中的排序可减少由于排序内存不足而造成的 I/O 开销。

28.7　本章小结

本章详细介绍了 SQL 语句的分析方法以及如何分析 SQL 语句的执行计划，通过分析执行计划可以发现某个 SQL 语句效率，以及是否会过多地占用系统的计算资源。在 Oracle 11g 中，虽然 SGA 的大部分内容由数据库服务器自动维护，但是仍然有些参数需要 DBA 参与，在 SGA 优化中，如 PRE_PAGE_SGA 的优化，使得在启动数据库实例时，将整个 SGA 读入物理内存，对于内存充足的系统而言，这样显然可以提高系统运行效率，使用 LOCK_SGA 优化，使得将 SGA 锁定（Lock）在物理内存内，这样就不会发生 SGA 使用虚拟内存的情况，从而提高数据的读取速度。在实际的应用中将程序和数据常驻内存，因为这样减少了物理读磁盘数据的次数，所以可以极大地提高数据的响应时间。

第 29 章

I/O以及系统优化

计算机的输入输出，即 I/O 是很耗时的系统行为，I/O 优化就是通过一定的措施减少 I/O 消耗的时间，从本质上讲，优化 I/O 的方法无非两种，一是最大化地减少 I/O 操作从而减少操作的数据量，二是平衡 I/O，如平衡数据库中各种文件的磁盘分布，这也同时减少了对统一磁盘文件的操作，从而减少了 I/O 量。本章讲述的 I/O 优化就是基于以上两点，只是针对不同粒度的数据库组件，优化的具体方法也不同。

29.1 I/O 优化

在处理 I/O 优化问题之前，首先需要确认优化的对象是什么，那么如何确认这个对象呢？答案是使用数据字典视图，所以在面临 Oracle 的 I/O 相关的性能问题时，应该使用视图 v$system_event、v$session_event、v$session_wait 视图来总和事件信息、会话信息以及等待事件相关的信息，提供与 I/O 相关的性能瓶颈。这里再次强调读者在实施 Oracle 数据库优化时，应该对常用的等待事件比较熟悉，这样对于等待事件就可以初步判断导致等待的原因，采取迭代的方式实现优化。

读者应注意一个问题，就是磁盘 I/O 的竞争是不可避免的，问题是这些磁盘 I/O 争用发生的频率，以及对于系统运行造成的影响，只有严重影响数据库系统性能的 I/O 竞争才能采取 I/O 优化措施。下面介绍不同的数据库组件的优化原则或具体方法。

29.1.1 表空间的 I/O 优化

在安装 Oracle 数据库时，所有表空间的数据文件都存放在相同的目录下，显然也都位于同一个物理磁盘上。这样的设置是不合理的，需要从 I/O 的角度进行优化，如将不同的表空间数据文件分布到不同的磁盘等。

Oracle 数据库在逻辑上划分成多个表空间，而表空间中包含物理的数据文件，这些文件是 Oracle 格式的操作系统文件，通过设计不同的表空间，从而存放和管理不同的数据库文件。表空间的分类如图 29-1 所示。

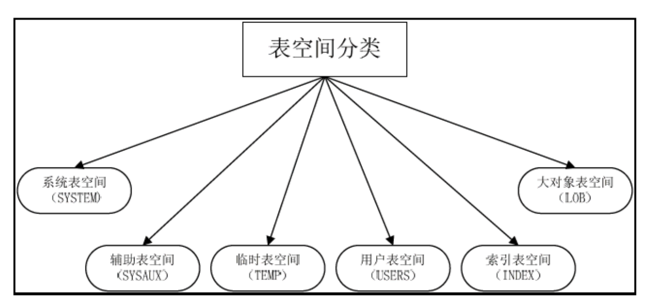


图 29-1　表空间的分类

1. 系统表空间

系统表空间主要用于存放数据库字典信息，但是在默认情况下用户数据也存储在系统表空间中，如果将用户数据和数据字典存放在相同的表空间显然存在磁盘 I/O 竞争，所以系统表空间中只存放和管理数据字典信息，对于用户数据可以创建用户表空间来单独存储。

2. 辅助表空间

辅助表空间名为 SYSAUX，是在 Oracle 数据库中为了减少 SYSTEM 系统表空间的信息而设计的表空间，通过系统表空间中存储的组建或功能分离出来，如存储 AWR（自动负载信息库）信息等。

3. 还原表空间

还原表空间主要用于回滚用户更改的数据，如用户删除一个大表时（DROP），此时会产生大量的还原数据，对还原表空间造成一定得维护压力，而且对于 OLTP 系统，用户会频繁地更改表数据，所以还原表空间会不断地填充数据和释放磁盘空间，这样就比较容易产生磁盘碎片，所以还原表空间应该只用于还原数据，而不要用作其他用途。

4. 临时表空间

临时表空间用于完成数据的磁盘排序行为，显然排序行为是数据库操作中比较频繁的一个数据库行为，这样就容易造成临时段的频繁分配和释放，也容易造成磁盘碎片，所以临时表空间不能存放永久数据，也不要存放（如索引等）其他数据库对象。

5. 用户表空间

实际的数据库系统肯定需要存储用户数据，主要是表数据，而表数据是应用程序或用户频繁操作的数据，所以需要单独建立一个表空间，关键是表空间中的数据文件要存储在单独的磁盘上，如果系统硬件资源不足，可以存储在 I/O 操作不频繁的磁盘上以减少 I/O 竞争产生的 I/O 等待。除了用户表空间外，对于大型的表还需要建立索引，而索引和用户表最好放在不同的磁盘驱动器上，显然使用表的索引与搜索表数据同样存在竞争，所以最好创建索引表空间，即用户表空间分为用户数据表空间和用户索引表空间，将二者的数据文件存储在不同的磁盘上，可以有效地减少磁盘 I/O，从而优化 I/O 行为。

6. 大对象表空间

大对象表空间是拥有大的数据库对象的表空间，该表空间的尺寸根据数据库块尺寸的不同而

有差异，分布范围为 8TB~128TB。Oracle 数据库在处理数据时都需要将表数据从数据文件所在磁盘读取到数据库高速缓冲区中，这个过程是机械行为，所以相对于内存行为要消耗更多的时间，对于大对象表空间尤其需要考虑 I/O 优化，对于大对象表空间应该使用更大的块尺寸，使得每次读取数据时可以获得比小的块尺寸更多的数据，从而减少磁盘 I/O。


说 明

上述对于表空间优化只给出了一个建议，总的思想是将不同表空间分布到不同的磁盘上，即将不同表空间的数据文件分开存放和管理，从而减少 I/O 竞争，对于具体的行为，如怎样将一个表空间的数据文件迁移到其他表空间、如何迁移系统表空间等操作可以参考本书的前面关于表空间管理一章的内容。

29.1.2 数据文件的 I/O 优化

在 Oracle 数据库中所有的数据都物理地存放在不同类型的数据库文件中，这些文件包括数据文件、数据文件、重做日志文件、控制文件、归档日志文件等，当后台进程访问这些数据时多个进程对同一个数据库文件的访问就造成 I/O 竞争，所以需要考虑对数据文件 I/O 优化的问题。

1. 数据库文件 I/O 优化

对数据库文件进行 I/O 优化时需要考虑的问题如下。

- 数据文件优化：存放用户实际使用的表数据或索引等数据库对象，尤其对于大型的数据库系统，必须创建单独的用户表空间而不能使用默认的系统表空间，并且用户表空间的数据文件应分布在和系统表空间不同的磁盘上，在用户表空间中可以创建多个数据文件，并将这些数据文件存放在不同的磁盘上以平衡磁盘 I/O 行为。
- 控制文件优化：控制文件是 Oracle 数据库中非常重要的数据库文件，在数据库启动时告诉数据库它的数据文件、重做日志文件等的目录位置和用于数据库实例恢复的信息，出于系统可靠性考虑，应该将控制文件分别存放在不同的磁盘上，通过冗余的方式保证控制文件的可靠性。这样的分布存放同时也实现了控制文件的磁盘 I/O。
- 重做日志文件优化：在 Oracle 数据库系统中，要求必须设置两个重做日志组，而且每个日志组中至少有一个重做日志成员，但是在生产数据库中应该创建不少于三个重做日志组，并且每个重做日志组不少于 2 个重做日志成员，这样可以保证重做日志文件的可靠性，以及提高数据库系统的效率（当然还要涉及重做日志文件的大小设置问题），Oracle 会循环使用重做日志组，在写一个重做日志组时，会同步写该组中的重做日志成员，所以对于同一个重做日志组中的重做日志成员应该存放在不同的磁盘上，以平衡数据 I/O。
- 重做日志文件和数据文件优化：重做日志文件和数据文件应该分开存放，即重做日志组中的每个成员和数据库系统中的数据文件应该分开存放，因为数据库写（DBWR）后台进程不但会操作数据文件，而且还会引发检查点使得重做日志写进程（LGWR）将数据库缓冲区中的脏数据同时写入重做日志文件，如果重做日志文件和数据文件放在同一个磁盘中，则存在 I/O 争用。
- 归档日志文件和重做日志文件优化：如果数据库处于归档模式，则重做日志文件在写满时切换到另一个重做日志组之前需要归档，即将当前重做日志文件中的数据全部写到归档文

件中，这样保证了介质恢复的数据要求，显然如果二者放在同一个磁盘下，在读出重做日志文件的同时要写入归档文件，存在 I/O 的争用，在实际的数据库系统中会造成重做日志切换时间过长，无法使用新的重做日志组而导致等待事件，所以应该将归档日志文件和重做日志文件分布到不同的磁盘上。

2. 数据库文件 I/O 监控

在实现了数据库文件的优化后，该如何监控数据库文件的 I/O 状况呢？即在数据库系统运行期间，如何通过某种方法知道当前数据库文件的信息，如输入输出量、读写消耗的时间等，答案是使用数据字典视图 v$filestat 实现。

先查看该数据字典视图的结构，然后分析一些重要的列属性，如实例 29-1 所示。

【实例 29-1】查看数据字典 v$filestat 的结构。

```
SQL> desc v$filestat;
 名称                                      是否为空？ 类型
 ------------------------------------------------------------- --
 FILE#                                        NUMBER
 PHYRDS                                       NUMBER
 PHYWRTS                                      NUMBER
 PHYBLKRD                                     NUMBER
 PHYBLKWRT                                    NUMBER
 SINGLEBLKRDS                                 NUMBER
 READTIM                                      NUMBER
 WRITETIM                                     NUMBER
 SINGLEBLKRDTIM                               NUMBER
 AVGIOTIM                                     NUMBER
 LSTIOTIM                                     NUMBER
 MINIOTIM                                     NUMBER
 MAXIORTM                                     NUMBER
 MAXIOWTM                                     NUMBER
```

从输出可以看出，该数据字典的表结构中列数据类型都为数字类型 NUMBER，下面分析几个重要的列属性。

- FILE：文件号说明对应的数据库文件标识，它和数据字典视图 dba_data_files 中的 file_id 含义相同，通过数据字典 dba_data_files 和 v$filestat 可以知道具体的一个数据文件的读取或写入统计数据。
- PHYRDS：说明物理读出该数据文件的数据块数。
- PHYWRTS：说明物理写入该数据文件的数据块数
- READTIM：说明读该数据文件使用的时间，单位是毫秒。
- WRITETIM：写入该数据文件使用的时间，单位是毫秒。

在使用数据字典 v$filestat 查看数据文件的操作统计数据前，需要设置系统参数 TIMED_STATISTICS 为 TRUE，这样数据字典 v$filestat 的 READTIM 和 WRITETIM 值才不会是 0，查看该参数的值，如实例 29-2 所示。

【实例 29-2】查看参数 TIMED_STATISTICS 的值。

```
SQL> show parameter timed_statistics;

NAME                                TYPE        VALUE
----------------------------------------------------------- -
timed_statistics                             boolean      FALSE
```

从输出可以看出该值为 FALSE，所以必须使用 ALTER SYSTEM 指令将该参数设置为 TRUE，如实例 29-3 所示。

【实例 29-3】设置参数 TIMED_STATISTICS 的值为 TRUE。

```
SQL> alter system set timed_statistics = true;

系统已更改。
```

显然，上述输出提示参数已经更改，接下来查询修改结果，如实例 29-4 所示。

【实例 29-4】查看参数 TIMED_STATISTICS 的值。

```
SQL> show parameter timed_statistics;

NAME                                TYPE        VALUE
----------------------------------------------------------
timed_statistics                            boolean     TRUE
```

现在可以使用数据字典 v$filestat 查看数据文件的 I/O 量信息了，如实例 29-5 所示。

【实例 29-5】使用数据字典 v$filestat 查看数据文件 I/O 量

```
SQL>  select file#,phyrds,phywrts,readtim,writetim
  2  from v$filestat
  3  order by file#;

    FILE#     PHYRDS    PHYWRTS    READTIM   WRITETIM
--------------------- ----------------- -------------------------
        1       5442        162         11572        52
        2         47        296         1511         90
        3        922        539         1339         39
        4         36          2        83            2
        5         10          2        73            2
```

从输出可以看出 FILE#为 1 的数据文件的物理读数据块数最多，显然其读操作所消耗的时间也最长，排在第二位的是 FILE#为 3 的数据文件，那么这两个数据文件到底存储在哪里？有什么作用呢？下面使用数据字典 dba_data_files 进行查看，它的 FILE_ID 和数据字典视图 v$filestat 中的 FILE#相同，如实例 29-6 所示。

【实例 29-6】查看数据库系统上的所有数据文件。

```
SQL> col file_id for 99
SQL> col file_name for a55
SQL> col tablespace_name for a15
```

```
SQL> select file_id,file_name,tablespace_name
  2  from dba_data_files
  3* order by file_id

FILE_ID FILE_NAME                                          TABLESPACE_NAME
------- ------------------------------------------------------------
1       F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSTEM01.DBF   SYSTEM
2       F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\UNDOTBS01.DBF  UNDOTBS1
3       F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\SYSAUX01.DBF   SYSAUX
4      F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\USERS01.DBF     USERS
5      F:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\EXAMPLE01.DBF   EXAMPLE
```

我们看到共有 5 个数据，文件号从 1~5，FILE_ID 为 1 的 SYSTEM01.DBF 文件是系统数据库文件，位于系统表空间（SYSTEM），而 FILE_ID 为 3 的 SYSAUS.DBF 文件是系统辅助数据库文件，位于辅助表空间（SYSAUX）。显然从文件 v$filestat 中得到的文件 I/O 量说明只有对于数据库系统表空间的操作，而很少有对数据文件的操作，因为 FILE#为 4 的数据文件地读写操作很少，而且该文件是 USER01.DBF 文件，是为了用户数据而创建的表空间。

通过使用这两个数据字典视图，可以追踪数据文件粒度的 I/O 操作，知道哪个数据文件发生了 I/O 量大的操作，如果 I/O 是由多个 I/O 量大的数据文件引起的，则需要平衡数据文件，将其分布到不同的磁盘上去。

29.1.3 表的 I/O 优化

对于表和索引的 I/O 优化，这里只给出一个优化思想，具体的操作可以参考创建表以及索引相关的章节。

如果已经将表和该表的索引存储在同一个表空间的数据文件中，则可以通过迁移表，将该表移动到另一个表空间中。下面通过实例 29-7~实例 29-10 说明如何将一个表迁移到另一个表空间。

【实例 29-7】查看 SCOTT 用户的 EMP 表的存储信息。

```
SQL> connect scott/tiger@orcl
已连接。
SQL> select table_name,tablespace_name
  2  from user_tables
  3  where table name ='EMP';

TABLE_NAME                    TABLESPACE_NAME
--------------------------------------------------
EMP                             USERS
```

上述输出说明，当前表 EMP 的存储表空间为 USERS，这里不再创建新的表空间，而是使用 SYSTEM 表空间作为新的表空间，将 EMP 表迁移到 SYSTEM 表空间，如实例 29-8 所示。

【实例 29-8】移动表 EMP 到新的表空间。

```
SQL> alter table emp move tablespace system;

表已更改。
```

上述输出表明，已经将表 EMP 移动到表空间 SYSTEM 中，下面使用数据字典 USER_TABLES 来验证移动结果，如实例 29-9 所示。

【实例 29-9】通过数据字典 USER_TABLES 来查看表 EMP 的表空间信息。

```
SQL> select table_name,tablespace_name
  2  from user_tables
  3  where table_name ='EMP';

TABLE_NAME                          TABLESPACE_NAME
------------------------------------------------- ----------------
EMP                                  SYSTEM
```

显然，此时 EMP 表的存储表空间已经是 SYSTEM，说明已成功移动表 EMP 到新的表空间中。当然这里是为了演示方便，没有新建一个表空间，使用 SYSTEM 表空间代替，这显然是不合理的，所以下面将表 EMP 再次移动到 USERS 表空间，如实例 29-10 所示。

【实例 29-10】将表 EMP 移动到 USERS 表空间。

```
SQL> alter table emp move tablespace users;

表已更改。
```

29.1.4 索引的 I/O 优化

通常建立索引的目的是加快表中数据的检索速度，但是建立索引对规模比较小的表而言，效果是不明显的，而对于规模较大的表则不同，可以明显地提高表中数据的搜索速度，提高响应事件。

索引也是一个表记录或位图记录，Oracle 在使用索引时同样需要对索引进行扫描，从而了解需要的数据记录位置，再通过 ROWID 来找到实际需要的表中的相关记录，但是在大型的 OLPT 数据库系统中，数据会频繁地插入或删除，如果删除的数据量很大，而用户又不断地搜索数据，此时就需要考虑对索引的 I/O 进行优化，原因是删除记录对应的索引并不会被 Oracle 删除而只是打上一个标记，这样造成索引记录中有大量的标记，使得查询有效索引记录的时间增加，考虑一个极限情况是：一个 10000 行记录的索引，删除了其中 9900 行索引记录，为了使用剩余的 100 行有效地索引记录，最糟糕的情况下需要扫描 10000 行索引记录，显然这降低了索引的使用效率，增加了索引操作的 I/O 数据量。此时为了避免这个问题需要对索引进行重建。查询 SCOTT 用户的 EMP 表上的索引如实例 29-11 所示。

【实例 29-11】查看表 EMP 上的索引。

```
SQL> col owner for a10
SQL> col index_name for a29
SQL> col table_name for a10
SQL> set line 100
SQL> select owner,index_name,index_type,table_name
  2  from dba_indexes
  3  where owner ='SCOTT'
  4* and table_name ='EMP'
```

```
OWNER      INDEX_NAME           INDEX_TYPE          TABLE_NAME
------------------ ------------------------------------- --------- -------
SCOTT      PK_EMP                  NORMAL                      EMP
SCOTT      SCOTT_EMP_INCOME_IDX  FUNCTION-BASED NORMAL  EMP
```

我们看到在 SCOTT 用户的 EMP 表上有一个基于函数的索引和一个主键索引，假设基于函数的索引 SCOTT_EMP_INCOME_IDX 因为被删除了大量的索引记录而造成了使用索引的 I/O 性能下降。首先需要进行索引的统计分析，如实例 29-12 所示。

【实例 29-12】使用 ANALYZE 指令分析索引。

```
SQL> analyze index scott.scott_emp_income_idx compute statistics;

索引已分析
```

然后，使用数据字典 INDEX_STATS 来查看删除索引相关的统计信息，如实例 29-13 所示。

【实例 29-13】使用数据字典 INDEX_STATS 来查看删除索引相关的统计信息。

```
SQL> select name,del_lf_rows,lf_rows_len
  2  from index_stats;

未选定行
```

因为索引 SCOTT_EMP_INCOME_IDX 中没有索引删除的记录，所以显示“未选定行”，数据字典 INDEX_STATS 中 NAME 的含义是索引的名字，DEL_IF_ROWS 的含义是删除的索引记录的长度，IF_ROWS_LEN 参数的含义是所有索引记录的长度。如果 DEL_IF_ROWS 占用 IF_ROWS_LEN 的 20%以上，则需要考虑重建索引，如实例 29-14 所示。

【实例 29-14】重建索引 SCOTT_EMP_INCOME_IDX。

```
SQL> alter index scott.scott_emp_income_idx rebuild;

索引已更改。
```

当使用 B-树索引时，如果索引的深度大于 3 也会影响使用索引的效率，此时也需要重建索引，先介绍如何查看索引的深度等信息，答案仍然是使用数据字典 DBA_INDEXES，如实例 29-15 所示。

【实例 29-15】查看索引的树深度信息。

```
SQL> select index_name,num_rows,tablespace_name,blevel,status
  2  from dba_indexes
  3* where table_name ='EMP'

INDEX_NAME            NUM_ROWS  TABLESPACE  BLEVEL  STATUS
--------------------------------------- ----------------------
PK_EMP                  12        USERS          0        VALID
SCOTT_EMP_INCOME_IDX    12        USERS          0        VALID
SCOTT_EMP_JOB_IDX        12        SYSTEM         0       UNUSABLE
```

输出显示结果中树的深度为 0，显然这样的索引是不需要优化的，如果通过查询认为有问题的索引，发现树的深度大于 3，则使用 ALTER INDEX 指令重建索引。参数 STATUS 说明当前索引

的状态，VALID 说明该参数有效，如果该值为 INVALID，则也需要重建该索引。

29.1.5 还原段的优化

还原段的作用是用户发出 ROLLBACK 指令时还原用户对数据的更改，即如果用户执行了插入数据，则执行删除操作，执行更新数据，则修改回原始值，执行了删除操作，则重新插入删除的记录，这些要恢复的数据放在还原段中，Oracle 为每个事务在还原表空间中分配一个还原段。

在删除一个大表时，还原数据可能会占满还原段的磁盘空间，从而导致数据库挂起，所以必须采取一定的措施。可使用 TRUNCATE 指令删除表中的数据，但是保留表结构，此时不会产生还原数据，然后再使用 DROP 命令删除空表（表的结构）。先创建一个模拟表，假设该表是大数据量的表，如实例 29-16 所示。

【实例 29-16】创建一个欲删除的模拟表。

```
SQL> create table large_table
  2  as
  3  select *
  4  from scott.emp;

表已创建。
```

利用以下代码验证该表是否成功创建：

```
SQL> select table name,status
  2  from user_tables
  3  where table_name like 'LAR%';

TABLE NAME                    STATUS
-------------------------------------------------- ----------------
LARGE_TABLE                   VALID
```

显然，表 LARGE_TABLE 已创建成功，下面使用 TRUNCATE 删除该表中的数据，如实例 29-17 所示。

【实例 29-17】利用 TRUNCATE 删除表 LARGE_TABLE 中的数据。

```
SQL> truncate table large table;

表被截断。
```

此时，表 LARGE_TABLE 的表结构还存在，继续使用 DROP 命令删除该表，这样就可以从系统中彻底删除大表 LARGE_TABLE，而不对还原段产生任何压力，如实例 29-18 所示。

【实例 29-18】利用 DROP 删除表 LARGE_TABLE。

```
SQL> drop table large table;

表已删除。
```

此时，删除了表 LARGE_TABLE，该表的定义已经不存在了，如实例 29-19 所示。

【实例 29-19】查询表 LARGE_TABLE 的表结构。

```
SQL> desc large_table;
ERROR:
ORA-04043: 对象 large_table 不存在
```

显然，已经不存在数据库对象 LARGE_TABLE，即彻底删除了具有大数据量的表 LARGE_TABLE。

在商业数据库中有时需要删除大表中的一部分数据，而保留一部分认为有价值的数据，此时可以采用中介的方式实现，即将有用的数据拷贝入临时表，该表作为一个存储中介，然后 TRUNCATE 原表，再将临时表中的数据拷贝入原表，最后删除临时表，这样就不会产生大量的还原数据，并减少还原段的 I/O 量。

可以使用 v$filestat 来查看 FILE#为 2 的文件的 I/O 量，如实例 29-20 所示。

【实例 29-20】查看数据文件 I/O 量。

```
SQL> select file#,phyrds,phywrts,readtim,writetim
  2  from v$filestat
  3  order by file#;

     FILE#     PHYRDS    PHYWRTS    READTIM   WRITETIM
---------- ---------- ---------- ---------- ----------
         1       5696        295      11747         57
         2         47        443       1511         90
         3        967        793       1382         58
         4         38          2         84          2
         5         10          2         73          2
```

FILE#为 2 数据文件的 I/O 量几乎没有什么变化，这应该归功于 TRUNCATE，因为它是 DDL 语句，不产生还原段数据。

29.2　系统优化

Oracle 作为一个数据库系统是运行在操作系统上的，无论如何它不能脱离操作系统的环境，所以收集某个时间段中核心数据是有帮助的，如收集 CPU 的使用、检测系统 I/O 以及监控内存使用等。下面从两种操作系统平台开始介绍如何监控系统资源，以优化 Oracle 的当前 OS 瓶颈。

29.2.1　在 Windows 平台实现性能监控

在 Windows 平台上使用“任务管理器”来监控系统中运行的进程状态，以及系统的性能问题，性能包括 CPU 的使用、内存的使用率以及 CPU 的使用频率图，同时按住 Ctrl+Alt+Delete 组合键启动“任务管理器”，打开“进程”选项卡，如图 29-2 所示，显示了当前运行的进程以及占用的 CPU 和内存资源。打开“性能”选项卡，如图 29-3 所示，显示了 CPU 以及内存使用的动态图。

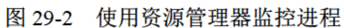



图 29-2　使用资源管理器监控进程

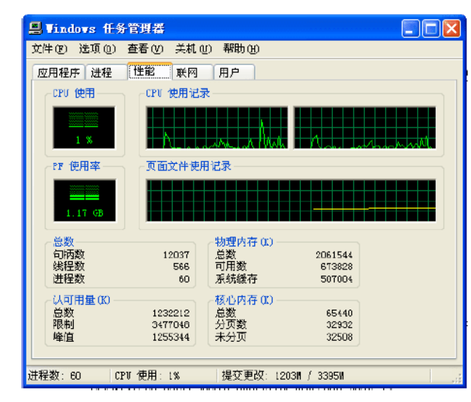


图 29-3　使用资源管理器监控性能

在使用资源管理器监控进程信息时，很容易确定哪个进程占用了最多的 CPU 以及占用了最多的内存等信息，从而定位哪些进程占用了宝贵的资源。一般在运行了 Oracle 的数据库服务器上，Oracle 是占用系统资源（CPU 和内存）最多的进程，Oracle 在 Windows 平台上显示为一个进程，而我们熟悉的实例后台进程在 Windows 平台上以线程的形式出现，这与 UNIX 系统不同。

由于 Windows 中监控 OS 性能的工具容易使用，直观简洁，所以我们不做过多介绍。

29.2.2　在 UNIX 系统上实现性能监控

下面介绍在 UNIX 系统上如何监控 CPU 的使用情况、设备使用情况以及虚拟内存的使用情况，最后简单介绍有关网络连接数据的统计信息。

1. 监控 CPU 的使用情况

在 Sun 的 Solaris 操作系统上可使用 SAR 指令监控 CPU 的使用情况，SAR 的意思是“系统活动报告”（System Activity Reporter），使用“-u”参数查看 CPU 的使用信息，先给出实例 29-21，然后解释其作用和相关参数的含义。

【实例 29-21】使用“sar –u”指令监控 CPU 的使用情况。

```
root@BJbackup # sar -u 5 10

SunOS BJbackup 5.6 Generic_105181-34 sun4u    09/29/09

16:23:23    %usr    %sys    %wio   %idle
16:23:33     29      10       2      63
16:23:38     15       9       2      74
16:23:43     14       8       1      77
16:23:48     14       9       1      75
16:23:53     14      10       1      75
16:23:58     11      10       0      78
16:24:03     16      10       1      73
16:24:08     16       9       2      73
16:24:13     14       9       0      77

Average      15       9       1      74
```

上述代码通过执行“sar –u 5 10”命令监控测量了 CPU 的使用情况，其中“-u”表示测量 CPU，第一个参数 5 表示测量周期，第二个参数 10 表示测量的次数，指令“sar –u 5 10”的含义就是每隔 5 秒钟测量一次 CPU 的使用情况，共检测 10 次。

下面解释 CPU 的使用情况，第一个参数“%usr”表示用户进程使用的 CPU 的百分比，在 Solaris 系统上 Oracle 服务器进程被认为是用户进程。第二个参数“%sys”表示操作系统自身消耗的 CPU 的百分比，如 CPU 的上下文环境切换、中断服务等。第三个参数“%wio”表示等待 CPU 的进程占用的 CPU 的百分比，显然这个数值越高说明系统效率越低。第四个参数“%idle”表示空闲 CPU 的百分比，说明这些 CPU 资源可用，从资源利用率的角度来看，此值越小越好，因为 CPU 资源得以充分利用。

注意，如果 CPU 的“%idle”值为“0%”并不意味着瓶颈出现，这要看等待使用 CPU 的队列长度，一般如果这个长度小于 CPU 个数的 2 倍，此时 CPU 的空闲率为 0%不说明系统存在 CPU 的资源瓶颈。下面使用“sar –q”指令查看执行队列的长度以及正在使用的 CPU 的百分比，如实例 29-22 所示。

【实例 29-22】使用“sar –q”查看 CPU 的使用情况。

```
root@BJbackup # sar -q 5 10

SunOS BJbackup 5.6 Generic_105181-34 sun4u    09/29/09

16:29:07 runq-sz %runocc swpq-sz %swpocc
16:29:12
16:29:17    1.0     20
16:29:22
16:29:27
16:29:32    1.0     20
16:29:37
16:29:42
16:29:47
16:29:52
16:29:57

Average     1.0      4
```

指令“sar –q 5 10”的含义是每 5 秒钟查看一次 CPU 的执行队列等信息，共测量 10 次，从上例的输出结果可以看出，在 10 次测量中不是每次各个参数都有数值。我们先分析参数的含义：第一个参数“runq-sz”说明当前 CPU 的执行队列中进程的数量，第二个参数“%runocc”说明正在运行的 CPU 占用的百分比，而其他两个参数“swpq-sz”和参数“%swpocc”表示交换队列的信息。

显然上述输出显示在测试时间段内，系统中 CPU 的执行队列中的进程数都为 1，在大多数情况下，根本没有排队，结合上例中的 CPU 的“%idle”数值说明当前系统的 CPU 资源不存在瓶颈，而且可以说 CPU 资源很充分。而正在运行的 CPU 平均只占 4%，也说明当前的 CPU 比较空闲。

2. 监控设备使用情况

遇到与系统 I/O 有关的瓶颈问题时，需要分析设备的使用情况，此时可使用“sar –d”指令测量关于设备的使用信息，如实例 29-23 所示。

【实例 29-23】使用“sar –d”测量和设备相关的信息。

```
Croot@BJbackup # sar -d 5 3

SunOS BJbackup 5.6 Generic 105181-34 sun4u    09/29/09

16:20:30   device       %busy   avque   r+w/s  blks/s  avwait  avserv

16:20:35   nfs1            0     0.0      0       0     0.0     0.0
          sd0            0     0.0      0       0     0.0     0.0
          sd0,a          0     0.0      0       0     0.0     0.0
          sd0,b          0     0.0      0       0     0.0     0.0
          sd0,c          0     0.0      0       0     0.0     0.0
          sd0,g          0     0.0      0       0     0.0     0.0
          sd0,h          0     0.0      0       0     0.0     0.0
          sd1            3     0.0      7     112     0.0     4.9
          sd1,a          3     0.0      7     112     0.0     4.9
          sd1,c          0     0.0      0       0     0.0     0.0
          sd6            0     0.0      0       0     0.0     0.0
..............为了节省篇幅，省略了两次测量结果。
Average    nfs1            0     0.0      0       0     0.0     0.0
          sd0            0     0.0      0       1     0.0    10.6
          sd0,a          0     0.0      0       1     0.0    10.6
          sd0,b          0     0.0      0       0     0.0     0.0
          sd0,c          0     0.0      0       0     0.0     0.0
          sd0,g          0     0.0      0       0     0.0     0.0
          sd0,h          0     0.0      0       0     0.0     0.0
          sd1            3     0.0      6      89     0.0     4.9
          sd1,a          3     0.0      6      89     0.0     4.9
          sd1,c          0     0.0      0       0     0.0     0.0
          sd6            0     0.0      0       0     0.0     0.0
```

指令“sar –d 5 3”表示测量三次设备的使用情况，每次间隔为 5 秒种，第一个 device 为设备名，第二个“%busy”为给定设备的繁忙百分比，第三个参数 avque 表示设备的平均队列长度，第四个参数“r+w/s”表示每秒该设备的读出和写入的数据，第五个参数“blks/s”表示该设备每秒传输的数据量，第六个参数 avwait 表示设备 I/O 操作所用的平均时间，第七个参数 avserv 表示 5 秒周期内每个 I/O 操作的平均等待时间。

在上述输出中应该注意“%busy”的值，如果该值超过 60%，一般表示该设备有问题，需要检测确认磁盘状况，当然平均队列长度、平均等待时间都是和系统的“%busy”相关的特性。还有就是 I/O 操作的平均使用时间不应该超过 100 毫秒，否则需要检测磁盘以找出原因。

3. 监控虚拟内存使用情况

我们使用 vmstat 指令获得虚拟内存的统计信息，如 vmstat 5 10，表示每 5 秒钟测试一次虚拟内存的使用，共测量 10 次，如实例 29-24 所示。

【实例 29-24】查看虚拟内存的使用信息。

```
root@BJbackup # vmstat 5 10
 procs     memory           page     disk          faults         cpu
```

```
 r b w   swap  free   re  mf pi  po fr de  sr s0 s1  s6 --   in  sy  cs  us sy id
 0 0 0   928  2920   0  0  0  0  0  0  0  0  5  0  0 4294967196 0 0 -14 -9 -103
 0 0 0 3446016 556288  0 1855 0  0  0  0  0  0  4  0  0  612 3159  578 16  9 76
 0 0 0 3445864 556496  0 1884 0  0  0  0  0  0  5  0  0  599 3102  549 12 10 78
 0 0 0 3446208 556432  0 1842 0  0  0  0  0  0  7  0  0  634 3272  602 15 10 76
 0 0 0 3446344 556472  0 1873 0  0  0  0  0  0  4  0  0  600 3149  561 14 10 77
 0 0 0 3445616 556392  0 1844 0  0  0  0  0  0  4  0  0  595 3027  536 12  9 79
 0 0 0 3445472 556208  0 1839 0  0  0  0  0  3 10  0  0  677 3377  590 20 10 70
 0 0 0 3446392 556568  0 1849 0  0  0  0  0  0  6  0  0  617 3206  574 14 10 77
 0 0 0 3446136 556320  0 1846 0  0  0  0  0  0  9  0  0  629 3135  555 13  9 79
 0 0 0 3446496 556576  0 1380 0  0  0  0  0  0  5  0  0  569 2452  464 10  8 83
 0 0 0 3446336 556472  0 1460 0  0  0  0  0  0 10  0  0  599 2972  477 12  8 80
```

虚拟内存状态信息的输出中包括 6 个部分，如下所示。

- PROCS：说明进程信息，其中 r 和 b 代表执行队列和阻塞队列的大小，r 值应该小于 2 倍的 CPU 大小，否则存在 CPU 瓶颈，CPU 无法正常处理排队的等待进程；b 说明阻塞队列的长度，此时的阻塞一般是由 I/O 引起，阻塞对象的长度可以及时地反映系统的性能。
- MEMORY：说明这些进程的内存使用状态，SWAP 说明当前可用的交换空间，单位是 K 字节，FREE 说明内存中自由表的大小，同样以 K 字节为单位。
- PAGE：说明进程换页信息，pi 和 po 说明调入内存页和调出内存页的字节数，单位为 K 字节。fr 说明空间的 K 字节数，de 说明短期内存不足的字节数，sr 说明由时钟扫描算法扫描的页面数，页面大小由操作系统确定，总之 si、so 提供页交换信息，一般如果没有过度分配 SGA 或 PGA 给数据库，则该值为 0，de 和 sr 以 K 字节为单位说明系统是否缺乏内存。
- DISK：说明磁盘使用信息，最好使用“sar –d”来查看更多的磁盘使用情况，
- fault：系统范围内的陷阱或故障信息，其中 in 表示设备中断数，sy 表示系统调用数，cs 表示 CPU 环境切换的数目。
- CPU：说明了 CPU 的使用信息，其中 us 说明用户进程使用 CPU 的百分比，sy 说明系统进程使用时间的百分比，id 说明非当前系统或用户进程使用的时间的百分比，如所有的 I/O 等待占用的 CPU 时间。

在 UNIX 系统的优化中，主要是使用相关的指令来查看系统的各种资源的使用情况，找出影响数据库运行效率的 OS 组件，如 CPU 瓶颈、系统内存不足等来优化数据库行为。

29.3　本章小结

在任何计算机上磁盘读非常占用系统时间，往往造成 CPU 等待或者系统应用的等待，平衡 I/O 是数据库优化的重要方面，在 Oracle 数据库中主要涉及表空间优化、数据文件优化、表优化、索引优化以及还原端优化，这些数据库对象的优化思想都是尽量减少磁盘读取的次数和平衡 I/O 数据量，从而减少磁盘读取造成的等待。

Oracle 数据库从宏观来讲就是一个应用软件，它安装在操作系统上，所以操作系统自身的性能无疑会影响数据库的性能，所以监视操作系统的性能以及解决操作系统的性能问题也是 DBA 优化数据库时需要认真对待的。